Separations between Combinatorial Measures for Transitive Functions

Sourav Chakraborty
Indian Statistical Institute, Kolkata, India

Chandrima Kayal
Indian Statistical Institute, Kolkata, India

Manaswi Paraashar
Indian Statistical Institute, Kolkata, India

Abstract

The role of symmetry in Boolean functions $f : \{0,1\}^n \to \{0,1\}$ has been extensively studied in complexity theory. For example, symmetric functions, that is, functions that are invariant under the action of S_n is an important class of functions in the study of Boolean functions. A function $f : \{0,1\}^n \to \{0,1\}$ is called transitive (or weakly-symmetric) if there exists a transitive group G of S_n such that f is invariant under the action of G. In other words, the value of a transitive function remains unchanged even after the input bits of f are moved around according to some permutation $\sigma \in \mathsf{G}$. Understanding various complexity measures of transitive functions has been a rich area of research for the past few decades.

In this work, we study transitive functions in light of several combinatorial measures. The question that we try to address in this paper is what is the maximum separations between various pairs of combinatorial measures for transitive functions. Such study for general Boolean functions has been going on for the past many years. The current best-known results for general Boolean functions have been nicely compiled by Aaronson et al. (STOC, 2021). But before this paper, no such systematic study has been done for the case of transitive functions.

The separation between a pair of combinatorial measures is shown by constructing interesting functions that demonstrate the separation. Over the past three decades, various interesting classes of functions have been designed for this purpose. In this context, one of the celebrated classes of functions is the "pointer functions". Ambainis et al. (JACM, 2017) constructed several functions, which are modifications of the pointer function in Göös et al. (SICOMP, 2018 / FOCS 2015), to demonstrate separation between various pairs of measures. In the last few years, pointer functions have been used to show separation between various other pairs of measures (Eg: Mukhopadhyay et al. (FSTTCS, 2015), Ben-David et al. (ITCS, 2017), Göös et al. (ToCT, 2018 / ICALP 2017)).

However, the pointer functions themselves are not transitive. Based on the various kinds of pointer functions, we construct new transitive functions whose deterministic query complexity, randomized query complexity, zero-error randomized query complexity, quantum query complexity, degree, and approximate degree are similar to that of the original pointer functions. Thus we demonstrate that even for transitive functions similar separations between pairs of combinatorial measures can be achieved by pointer functions.

Our construction of transitive functions depends crucially on the construction of particular classes of transitive groups whose actions, though involved, helps to preserve certain structural features of the input strings. The transitive groups we construct may be of independent interest in other areas of mathematics and theoretical computer science.

We summarize the current knowledge of relations between various combinatorial measures of transitive functions in a table similar to the table compiled by Aaronson et al. (STOC, 2021) for general functions.

2012 ACM Subject Classification Separations between Combinatorial Measures for Transitive Functions

Keywords and phrases Transitive functions, Combinatorial complexity of Boolean functions, Separation

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

For a Boolean function $f : \{0,1\}^n \to \{0,1\}$ what is the relationsip between its various combinatorial measures, like deterministic query complexity ($\mathsf{D}(f)$), bounded-error randomized and quantum query complexity ($\mathsf{R}(f)$ and $\mathsf{Q}(f)$ respectively), zero -randomized query complexity ($\mathsf{R}_0(f)$), exact quantum query complexity ($\mathsf{Q}_\mathsf{E}(f)$), sensitivity ($\mathsf{s}(f)$), block sensitivity ($\mathsf{bs}(f)$), certificate complexity ($\mathsf{C}(f)$), randomized certificate complexity ($\widetilde{\mathsf{RC}}(f)$), unambiguous certificate complexity ($\mathsf{UC}(f)$), degree ($\deg(f)$), approximate degree ($\widetilde{\deg}(f)$) and spectral sensitivity ($\lambda(f)$)[1]? For over three decades, understanding the relationships between these measures has been an active area of research in computational complexity theory. These combinatorial measures have applications in many other areas of theoretical computer science, and thus the above question takes a central position.

In the last couple of years, some of the more celebrated conjectures have been answered - like the quadratic relation between sensitivity and degree of Boolean functions [18]. We refer the reader to the survey [11] for an introduction to this area.

Understanding the relationship between various combinatorial measures involves two parts:

- Relationships - proving that one measure is upper bounded by a function of another measure. For example, for any Boolean function f, $\deg(f) \leq \mathsf{s}(f)^2$ and $\mathsf{D}(f) \leq \mathsf{R}(f)^2$.
- Separations - constructing functions that demonstrates separation between two measures. For example, there exists a Boolean function f with $\deg(f) \geq \mathsf{s}(f)^2$. Also there exists another Boolean function g with $\mathsf{D}(g) \geq \mathsf{R}(g)^2$.

Obtaining tight bounds between pairs of combinatorial measures - that is, when the relationship and the separation results match - is the holy grail of this area of research. The current best known results for different pairs of functions have been nicely compiled in [3].

For special classes of Booleans functions the relationships and the separation results might be different than that of general Boolean functions. For example, while it is known that there exists f such that $\mathsf{bs}(f) = \Theta(\mathsf{s}(f)^2)$ [28], for a symmetric function a more tighter result is known, $\mathsf{bs}(f) = \Theta(\mathsf{s}(f))$. The best known 'relationship' of $\mathsf{bs}(f)$ for a general Boolean functions is $\mathsf{s}(f)^4$ [18]. How the various measures behave for different classes of functions has been studied since the dawn of this area of research.

Transitive Functions: One of the most well-studied classes of Boolean functions is that of the transitive functions. A function $f : \{0,1\}^n \to \{0,1\}$ is transitive if there is a transitive group $G \leq \mathsf{S}_n$ such that the function value remains unchanged even after the indices of the input is acted upon by a permutation from G. Note that, when $G = \mathsf{S}_n$ then the function is symmetric. Transitive functions (also called "weakly symmetric" functions) has been studied extensively in the context of various complexity measure. This is because symmetry is a natural measure of the complexity of a Boolean function. It is expected that functions with more symmetry must have less variation among the various combinatorial measures. A recent work [7] has studied the functions under various types of symmetry in terms of quantum speedup. So, studying functions in terms of symmetry is important in various aspects.

For example, for symmetric functions, where the transitive group is S_n, most of the

[1] We provide formal definitions of the measures used in this paper in Appendix B.

combinatorial measures become the same up to a constant [2]. Another example of transitive functions is the graph properties. The input is the adjacency matrix, and the transitive group is the graph isomorphism group acting on the bits of the adjacency matrix. [34, 31, 22, 14] tried to obtain tight bounds on the relationship between sensitivity and block sensitivity for graph properties. They also tried to answer how low can sensitivity and block sensitivity go for graph properties?

In papers like [32, 12, 30, 13] it has been studied how low can the combinatorial measures go for transitive functions. The behavior of transitive functions can be very different from general Boolean functions. For example, while it is known that there are Boolean functions for which the sensitivity is as low as $\Theta(\log n)$ where n is the number of effective variables[3], it is known (from [30] and [18]) that if f is a transitive function on n effective variables then its sensitivity $s(f)$ is at least $\Omega(n^{1/12})$[4]. Similar behavior can be observed in other measures too. For example, it is easy to see that for a transitive function the certificate complexity is $\Omega(\sqrt{n})$, while the certificate complexity for a general Boolean function can be as low as $O(\log n)$. In Table 3 we summarize the best-known separations of the combinatorial measures for transitive functions. A natural related question is:

What is the tight relationship between various pairs of combinatorial measures for transitive functions?

By definition, the known 'relationship' results for general functions hold for transitive functions. But tighter 'relationship' results may be obtained for transitive functions. On the other hand, the existing 'separations' doesn't extend easily since the example used to demonstrate separation between certain pairs of measures may not be transitive. Some of the most celebrated examples are not transitive. For example some of the celebrated function construction like the pointer function [5], used for demonstrating tight separations between various pairs like $\mathsf{D}(f)$ and $\mathsf{R}_0(f)$, are not transitive. Similarly, the functions constructed using the cheat sheet techniques [2] used for separation between quantum query complexity and degree, or approximate degree are not transitive. Constructing transitive functions which demonstrate tight separations between various pairs of combinatorial measures is very challenging.

Our Results: We try to answer the above question for various pairs of measures. More precisely, our main contribution is to construct transitive functions that have similar complexity measures as the *pointer functions*. Hence for those pairs of measures where pointer functions can demonstrate separation for general functions, we prove that similar separation can also be demonstrated by transitive functions.

Our results and the current known relations between various pairs of complexity measures of transitive functions are compiled in Table 1. This table is along the lines of the table in [3] where the best-known relations between various complexity measures of genral Boolean functions were presented.

Deterministic query complexity and zero-error randomized query complexity are two of the most basic measures and yet the tight relation between these measures was not known until recently. In [29] they showed that for the "balanced NAND-tree" function, $\widetilde{\wedge}$-tree, $\mathsf{D}(\widetilde{\wedge}\text{-tree}) \geq \mathsf{R}_0(\widetilde{\wedge}\text{-tree})^{1.33}$. Although the function $\widetilde{\wedge}$-tree is transitive, the best

[2] There are still open problems on the tightness of the constants.
[3] A variable is effective if the function is dependent on it.
[4] It is conjectured that the sensitivity of a transitive function is $\Omega(n^{1/3})$.

known 'relationship' was quadratic, that is for all Boolean function f, $\mathsf{D}(f) = O(\mathsf{R_0}(f)^2)$. In [5] a new function, $\mathsf{A1}$, was constructed for which deterministic query complexity and zero-error randomized query complexity can have a quadratic separation between them, and this matched the known 'relationship' results. The function in [5] was a variant of the pointer functions - a class of functions introduced by [17] that has found extensive usage in showing separations between various complexity measures of Boolean functions. The function, $\mathsf{A1}$, also gave (the current best known) separations between deterministic query complexity and other measures like quantum query complexity and degree. But the function $\mathsf{A1}$ is not transitive. Using the $\mathsf{A1}$ function we construct a transitive function that demonstrates a similar gap between deterministic query complexity and zero-error randomized query complexity, quantum query complexity, and degree.

▶ **Theorem 1.** *There exists a transitive function F_1 such that*

$$\mathsf{D}(F_1) = \widetilde{\Omega}(\mathsf{Q}(F_1)^4), \qquad \mathsf{D}(F_1) = \widetilde{\Omega}(\mathsf{R_0}(F_1)^2), \qquad \mathsf{D}(F_1) = \widetilde{\Omega}(\deg(F_1)^2).$$

In [5, 8] various variants of the pointer function have been used to show separation between other pairs of measures like $\mathsf{R_0}$ with R, $\mathsf{Q_E}$, deg, and Q, R with $\widetilde{\deg}$, deg, $\mathsf{Q_E}$ and sensitivity. Inspired from these function we construct transitive versions that demonstrate similar separation for transitive functions as that of general functions. The construction of these functions, though more complicated and involved, are similar in flavor to that of F_1. The proof of Theorem 1 is presented in Section 4. The formal statements of our other results and their proofs are presented in the Appendix D, E and F. Our proof techniques also help us make transitive versions of other functions like that used in[2] to demonstrate the gap between Q and certificate complexity. The result is presented in Appendix G. All our results are compiled (and marked in green) in Table 1.

One would naturally ask what stops us from constructing transitive functions analogous to the other functions, like cheat sheet-based functions. In fact, one could ask why to use ad-hoc techniques to construct transitive functions (as we have done in most of our proofs) and instead why not design a unifying technique for converting any function into a transitive function that would display similar properties in terms of combinatorial measures [5]. If one could do so, all the 'separation' results for general functions (in terms of separation between pairs of measures) would translate to separation for transitive functions. In Appendix I we have discussed why such a task is challenging. We argue the challenges of making transitive versions of the cheat-sheet functions.

2 Notations and Background

2.1 Notations and basic definitions

We use $[n]$ to denote the set $\{1, \ldots, n\}$. $\{0,1\}^n$ denotes the set of all n-bit binary strings. For any $X \in \{0,1\}^n$ the Hamming Weight of X (denoted $|X|$) will refer to the number of 1 in X. 0^n and 1^n denotes all 0's string of n-bit and all 1's string of n-bit, respectively.

We denote by S_n the set of all permutations on $[n]$. Given an element $\sigma \in \mathsf{S}_n$ and a n-bit string $x_1, \ldots, x_n \in \{0,1\}^n$ we denote by $\sigma[x_1, \ldots, x_n]$ the string obtained by permuting the

[5] In [7] they have demonstrated a technique that can be used for constructing a transitive partial function that demonstrates gaps (between certain combinatorial measures) similar to a given partial function that need not be transitive. But their construction need not construct a total function even when the given function is total.

Table 1 Best known separations between combinatorial measures for transitive functions.

	D	R_0	R	C	RC	bs	s	λ	Q_E	deg	Q	$\widetilde{deg}$
D		2 ; 2 T:1	2 ; 3 T:1	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:91	4 ; 6 T:73	2 ; 3 T:69	2 ; 3 T:1	4 ; 4 T:1	4 ; 4 T:73
R_0	1,1 $\oplus$		2 ; 2 T:69	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:91	4 ; 6 T:73	2 ; 3 T:69	2 ; 3 T:69	3 ; 4 T:74	4 ; 4 T:73
R	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:91	4 ; 6 T:73	1.5 ; 3 T:75	2 ; 3 T:73	2 ; 4 $\wedge$	4 ; 4 T:73
C	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 2 $\oplus$		2 ; 2 [15]	2 ; 2 [15]	2 ; 5 [28]	2 ; 6 $\wedge$	1.15 ; 3 [4]	1.63 ; 3 [26]	2 ; 4 $\wedge$	2 ; 4 $\wedge$
RC	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		1.5 ; 2 [15]	2 ; 4 [28]	2 ; 4 $\wedge$	1.15 ; 2 [4]	1.63 ; 2 [26]	2 ; 2 $\wedge$	2 ; 2 $\wedge$
bs	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 4 [28]	2 ; 4 $\wedge$	1.15 ; 2 [4]	1.63 ; 2 [26]	2, 2 $\wedge$	2 ; 2 $\wedge$
s	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 2 $\wedge$	1.15 ; 2 [4]	1.63 ; 2 [26]	2, 2 $\wedge$	2 ; 2 $\wedge$
λ	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$
Q_E	1 ; 1 $\oplus$	1.33 ; 2 $\widetilde{\wedge}$-tree	1.33 ; 3 $\widetilde{\wedge}$-tree	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 6 T:95	2 ; 6 T:95		1 ; 3 $\oplus$	2 ; 4 $\wedge$	1 ; 4 $\oplus$
deg	1 ; 1 $\oplus$	1.33 ; 2 $\widetilde{\wedge}$-tree	1.33 ; 2 $\widetilde{\wedge}$-tree	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge$	1 ; 1 $\oplus$		2 ; 2 $\wedge$	2 ; 2 $\wedge$
Q	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	2 ; 2 T:95	2 ; 3 T:95	2 ; 3 T:95	2 ; 6 T:95	2 ; 6 T:95	1, 1 $\oplus$	1 ; 3 $\oplus$		1 ; 4 $\oplus$
$\widetilde{deg}$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	

)
- Entry $a; b$ in row A and column B represents: (1) ('Relationships') for any Boolean function f , $A(f) = O(B(f))^{b+o(1)}$, and (2) ('Separations') there exists a *transitive* function g such that $A(g) = \Omega(B(g))^a$.
- Cells with a green background are those for which we constructed new *transitive functions* to demonstrate 'separations' that match the known 'separations' for general functions. The previously known functions that gave the strongest 'separations' were not transitive. The second row gives the reference to the Theorems where the 'separation' is proved. Although for these green cells the bounds match that of the general functions, for some cells (with light green color) there is a gap between the known 'relationships' and 'separations'.
- In the cells a with white background the best-known examples for the corresponding separation was already *transitive functions*. For these cells, the second row either contains the function that demonstrates the separation or a reference to the paper where the separation was proved. So for these cells, the 'separations' for transitive functions matched the current best known 'separations' for general functions. Note that for some of these cell the don't match.
- Cells with a yellow background are those where the 'separations' do not match the best known 'separations' for general functions.
- The various functions mentioned/referred in this table are defined in Appendix C.

indices according to σ. That is $\sigma[x_1, \ldots, x_n] = x_{\sigma(1)}, \ldots, x_{\sigma(n)}$. This is also called the action of σ on the $x_1, \ldots, x_n$.

Following are a couple of interesting elements of S_n that will be used in this paper.

▶ **Definition 2.** *For any $n = 2k$ the* flip *swaps $(2i-1)$ and $2i$ for all $1 \leq i \leq k$. The permutation $\mathsf{Swap}_{\frac{1}{2}}$ swaps i with $(k+i)$, for all $1 \leq i \leq k$. That is,*

$$\mathsf{flip} = (1,2)(3,4)\ldots(n-1,n) \qquad \& \qquad \mathsf{Swap}_{\frac{1}{2}}[x_1, \ldots, x_{2k}] = x_{k+1}, \ldots, x_{2k}, x_1 \ldots, x_k.$$

Every integer $\ell \in [n]$ has the canonical $\log n$ bit string representation. However the number of 1's and 0's in such a representation is not same for all $\ell \in [n]$. The following representation of $\ell \in [n]$ ensures that for all $\ell \in [n]$ the encoding has same Hamming weight.

▶ **Definition 3** (Balanced binary representation)**.** *For any $\ell \in [n]$, let $\ell_1, \ldots, \ell_{\log n}$ be the binary representation of the number ℓ where $\ell_i \in \{0,1\}$ for all i. Replacing 1 by 10 and 0 by 01 in the binary representation of ℓ, we get a $2\log n$-bit unique representation, which we call* Balanced binary representation *of ℓ and denote as $bb(\ell)$.*

In this paper all the functions considered are of form $F : \{0,1\}^n \to \{0,1\}^k$. By Boolean functions we would mean a Boolean valued function that is of the form $f : \{0,1\}^n \to \{0,1\}$.

An input to a function $F : \{0,1\}^n \to \{0,1\}^k$ is a n-bit string but also the input can be thought of as different objects. For example, if the $n = NM$ then the input may be thought of as a $(N \times M)$-matrix with Boolean values. It may also be thought of as a $(M \times N)$-matrix.

If $\Sigma = \{0,1\}^k$ then $\Sigma^{(n \times m)}$ denotes an $(n \times m)$-matrix with an element of Σ (that is, a k-bit string) stored in each cell of the matrix. Note that $\Sigma^{(n \times m)}$ is actually $\{0,1\}^{mnk}$. Thus, a function $F : \Sigma^{(n \times m)} \to \{0,1\}$ is actually a Boolean function from a $\{0,1\}^{nmk}$ to $\{0,1\}$, where we think of the input as an $(n \times m)$-matrix over the alphabet Σ.

One particular nomenclature that we use in this paper is that of 1-cell certificate.

▶ **Definition 4** (1-cell certificate)**.** *Given a function $f : \Sigma^{(n \times m)} \to \{0,1\}$ (where $\Sigma = \{0,1\}^k$) the 1-cell certificate is a partial assignments to the cells which forces the value of the function to 1. So a 1-cell certificate is of the form $(\Sigma \cup \{*\})^{(n \times m)}$. Note the here we assume that the contents in any cell is either empty or a proper element of Σ (and not a partial k-bit string).*

Another notation that is often used is the following:

▶ **Notation 5.** *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are groups on $[n]$ and $[m]$ then the group $A \times B$ act on the cells on the matrix. Thus for any $(\sigma, \sigma') \in A \times B$ and a $M \in \Sigma^{(n \times m)}$ by $(\sigma, \sigma')[M]$ we would mean the permutation on the cell of M according to (σ, σ') and move the contains in the cells accordingly. Note that the relative position of bits within the contents in each cell is not touched.*

Next, we define the composition of two Boolean functions.

▶ **Definition 6** (Composition of functions)**.** *Let $f : \{0,1\}^{nk} \to \{0,1\}$ and $g : \{0,1\}^m \to \{0,1\}^k$ be two functions. The* composition of f and g, *denoted by $f \circ g : \{0,1\}^{nm} \to \{0,1\}$, is defined to be a function on nm bits such that on input $x = (x_1, \ldots, x_n) \in \{0,1\}^{nm}$, where each $x_i \in \{0,1\}^m$, $f \circ g(x_1, \ldots, x_n) = f(g(x_1), \ldots, g(x_n))$. We will refer f as* outer function *and g as* inner function.

2.2 Transitive Groups and Transitive Functions

The central objects in this paper are transitive Boolean function. We first define transitive groups.

▶ **Definition 7.** *A group $G \leq \mathsf{S}_n$ is transitive if for all $i, j \in [n]$ there exists a $\sigma \in G$ such that $\sigma(i) = j$.*

▶ **Definition 8.** *For $f : A^n \to \{0,1\}$ and $G \leq \mathsf{S}_n$ we say f is invariant under the action of G, if for all $\alpha_1, \ldots, \alpha_n \in A$.*

$$f(\alpha_1, \ldots, \alpha_n) = f(\alpha_{\sigma(1)}, \ldots, \alpha_{\sigma(n)}).$$

▶ **Observation 9.** *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are transitive groups groups on $[n]$ and $[m]$ then the group $A \times B$ is a transitive group acting on the cells on the matrix.*

There are many interesting transitive groups. The symmetric group is indeed transitive. The graph isomorphism group (that acts on the adjacency matrix - minus the diagonal - of a graph by changing the ordering on the vertices) is transitive. The cyclic permutation over all the points in the set is a transitive group. The following is another non-trivial transitive group on $[k]$ that we will use extensively in this paper.

▶ **Definition 10.** *For any k that is a power of 2, the Binary-tree-transitive group Bt_k is a subgroup of S_k. To describe its generating set we think of group Bt_k acting on the elements $\{1, \ldots, k\}$ and the elements are placed in the leaves of a balanced binary tree of depth $\log k$ - one element in each leaf. Each internal node (including the root) corresponds to an element in the generating set of Bt_k. The element corresponding to an internal node in the binary tree swaps the left and right sub-tree of the node. The permutation element corresponding to the root node is called the Root-swap as it swaps the left and right sub-tree to the root of the binary tree.*

We now state two claims whose proofs are given in Appendix B.1.

▷ Claim 11. The group Bt_k is a transitive group.

The following claim describes how the group Bt_k acts on various encoding of integers. Recall the balance-binary representation (Definition 3).

▷ Claim 12. For all $\widehat{\gamma} \in \mathsf{Bt}_{2\log n}$ there is a $\gamma \in \mathsf{S}_n$ such that for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

Now let us consider another encoding that we will using for the set of rows and columns of a matrix.

▶ **Definition 13.** *Given a set R of n rows $r_1, \ldots, r_n$ and a set C of n columns $c_1, \ldots, c_n$ we define the balanced-pointer-encoding function $\mathcal{E} : (R \times \{0\}) \cup (\{0\} \times C) \to \{0,1\}^{4\log n}$, as follows:*

$$\mathcal{E}(r_i, 0) = bb(i) \cdot 0^{2\log n}, \text{ and, } \mathcal{E}(0, c_j) = 0^{2\log n} \cdot bb(j).$$

The following is a claim is easy to verify.

▷ Claim 14. Let R be a set of n rows $r_1, \ldots, r_n$ and C be a set of n columns $c_1, \ldots, c_n$ and consider the *balanced-pointer-encoding function* $\mathcal{E} : (R \times \{0\}) \cup (\{0\} \times C) \to \{0,1\}^{4\log n}$. For

any elementary permutation $\widehat{\sigma}$ in $\mathsf{Bt}_{4\log n}$ (other than the Root-swap) there is a $\sigma \in \mathsf{S}_n$ such that for any $(r_i, c_j) \in (R \times \{0\}) \cup (\{0\} \times C)$

$$\widehat{\sigma}[\mathcal{E}(r_i, c_j)] = \mathcal{E}(r_{\sigma(i)}, c_{\sigma(j)}),$$

where we assume $r_0 = c_0 = 0$ and any permutation of in S_n sends 0 to 0.

If $\widehat{\sigma}$ is the root-swap then for any $(r_i, c_j) \in (R \times \{0\}) \cup (\{0\} \times C)$

$$\widehat{\sigma}[\mathcal{E}(r_i, c_j)] = \mathsf{Swap}_{\frac{1}{2}}(\mathcal{E}(r_i, c_j)) = \mathcal{E}(c_j, r_i).$$

2.3 Pointer function

For the sake of completeness first we will describe the function introduced in [5] that achieves separation between several complexity complexity measures like *Deterministic query complexity*, *Randomized query complexity*, *Quantum query complexity* etc. This function was originally motivated from [17] function. There are three three variants of [5] function that have some special kind of non-Boolean domain, which we call *Pointer matrix*. Our function is a special "encoding" of that non-Boolean domain such that the resulting function becomes transitive and achieves the separation between complexity measures that matches the known separation between the general functions. Here we will define only the first variant of [5] function. Rest of the variants are defined in Appendix C.1.

▶ **Definition 15** (Pointer matrix over Σ). *For $m, n \in \mathbb{N}$, let M be a $(m \times n)$ matrix with m rows and n columns. We refer to each of the $m \times n$ entries of M as* cells. *Each cell of the matrix is from a alphabet set Σ where $\Sigma = \{0,1\} \times \widetilde{P} \times \widetilde{P} \times \widetilde{P}$ and $\widetilde{P} = \{(i,j) | i \in [m], j \in [n]\} \cup \{\perp\}$. We call $\widetilde{P}$ as set of pointers where, pointers of the form $\{(i,j) | i \in [m], j \in [n]\}$ pointing to the cell (i,j) and $\perp$ is the null pointer. Hence,each entry $x_{(i,j)}$ of the matrix M is a 4-tuple from Σ. The elements of the 4-tuple we refer as* value, left pointer, right pointer *and* back pointer *respectively and denote by Value$(x_{(i,j)})$, LPointer$(x_{(i,j)})$, RPointer$(x_{(i,j)})$ and BPointer$(x_{(i,j)})$ respectively where Value $\in \{0,1\}$, LPointer, RPointer, BPointer $\in \widetilde{P}$. We call this type of matrix as* pointer matrix *and denote by $\Sigma^{n \times n}$.*

A special case of the pointer-matrix, which we call Type_1 pointer matrix over Σ, *is when for each cell of M, BPointer $\in \{[n] \cup \perp\}$ that is backpointers are pointing to the columns of the matrix.*

Also, in general when, BPointer $\in \{(i,j) | i \in [m], j \in [n]\} \cup \{\perp\}$, we call it a Type_2 pointer matrix over Σ.

Now we will define some additional properties of the domain that we need to define [5] function.

▶ **Definition 16** (Pointer matrix with marked column). *Let M be an $m \times n$* pointer-matrix *over Σ. A column $j \in [n]$ of M is defined to be a* marked column *if there exists exactly one cell (i,j), $i \in [m]$, in that column with entry $x_{(i,j)}$ such that $x_{(i,j)} \neq (1, \perp, \perp, \perp)$ and every other cell in that column is of the form $(1, \perp, \perp, \perp)$. The cell (i,j) is defined to be the* special element *of the* marked column *j.*

Let n be a power of 2. Let T be a rooted, directed and balanced binary tree with n-leaves and $(n-1)$ internal vertices. We will use the following notations that will be used in defining some functions formally.

▶ **Notation 17.** *Let n be a power of 2. Let T be a rooted, directed and balanced binary tree with n-leaves and $(n-1)$ internal vertices. Labels the edges of T as follows: the outgoing*

edges from each node are labeled by either left or right. The leaves of the tree are labeled by the elements of $[n]$ from left to right, with each label used exactly once. For each leaf $j \in [n]$ of the tree, the path from the root to the leaf j defines a sequence of left and right of length $O(\log n)$, which we denote by $T(j)$.

When n is not a power of 2, choose the largest $k \in \mathbb{N}$ such that $2^k \leq n$, consider a complete balanced tree with 2^k leaves and add a pair of child node to to each $n - 2^k$ leaves starting from left. Define $T(j)$ as before.

More details about *partial assignment* and *certificates* can be found in Definition 35. Now we are ready to describe the *Variant 1* of [5] function.

▶ **Definition 18** (Variant 1 [5]). *Let $\Sigma^{m\times n}$ be a* Type_1 pointer matrix *where BPointer is a pointer of the form $\{j|j \in [n]\}$ that points to other column and LPointer, RPointer are as usual points to other cell. Define $\mathsf{A1}_{(m,n)} : \Sigma^{m\times n} \to \{0, 1\}$ on a* Type_1 pointer matrix *such that for all $x = (x_{i,j}) \in \Sigma^{m\times n}$, the function $\mathsf{A1}_{(m,n)}(x_{i,j})$ evaluates to 1 if and only if it has a* 1- cell certificate *of the following form:*

1. *there exists exactly one* marked column *$j^\star$ in M,*
2. *There is a special cell, say $(i^\star, j^\star)$ which we call the special element in the the* marked column *$j^\star$ and there is a balanced binary tree T rooted at the special cell,*
3. *for each non-marked column $j \in [n] \setminus \{j^\star\}$ there exist a* cell *l_j such that $Value(l_j) = 0$ and $BPointer(l_j) = j^\star$ where l_j is the end of the path that starts at the* special element *and follows the pointers LPointer and RPointer as specified by the sequence $T(j)$. l_j exists for all $j \in [n] \setminus \{j^\star\}$ i.e. no pointer on the path is $\perp$. We refer l_j as the leaves of the tree.*

The above function achieves the separation between D vs. $\mathsf{R_0}$ and D vs. Q for $m = 2n$. Here we will restate some of the results from [5] which we will use to prove the results for our function:

▶ **Theorem 19** ([5]). *The function $\mathsf{A1}_{(m.n)}$ in Definition 18 satisfies*

$\mathsf{D} = \Omega(n^2)$ *for $m = 2n$ where $m, n \in \mathbb{N}$,*

$\mathsf{R_0} = \widetilde{O}(m + n)$ *for any $m, n \in \mathbb{N}$,*

$\mathsf{Q} = \widetilde{O}(\sqrt{m} + \sqrt{n})$ *for any $m, n \in \mathbb{N}$.*

Though [5] gives the deterministic lower bound for the function $\mathsf{A1}$ precisely for $2m \times m$ matrices following the same line of argument it can be proved that $\mathsf{D}(\Omega(n^2))$ holds for $n \times n$ matrices also. For sake of completeness we give a proof for $n \times n$ matrices.

▶ **Theorem 20.** $\mathsf{D}(\mathsf{A1}_{(n,n)}) = \Omega(n^2)$.

Adversary Strategy for $\mathsf{A1}_{(n,n)}$: We describe an adversary strategy that ensures that the value of the function is undetermined after $\Omega(n^2)$ queries. Assume that deterministic query algorithm queries a cell (i, j). Let k be the number of queried cell in the column j. If $k \leq \frac{n}{2}$ adversary will return $(1, \perp, \perp, \perp)$. Otherwise adversary will return $(0, \perp, \perp, n - k)$.

▷ Claim 21. The value of the function $\mathsf{A1}_{(n,n)}$ will be undetermined if there is a column with at most $n/2$ queried cells in the first $\frac{n}{2}$ columns $\{1, 2, \ldots, \frac{n}{2}\}$ and at least $3n$ unqueried cells in total.

Proof. Adversary can always set the value of function to 0 if the conditions of the claim are satisfied.

Adversary can also set the value of the function to 1**:** If $s \in [\frac{n}{2}]$ be the column with at most $\frac{n}{2}$ queried cell, then all the queried cells of the column are of the form $(1, \perp, \perp, \perp)$. Assign $(1, \perp, \perp, \perp)$ to the other cell and leave one cell for the *special element* $a_{p,s}$(say).

For each non-marked column $j \in [n]$ s define l_j as follows: If column j has one unqueried cell then assign $(0, \perp, \perp, s)$ to that cell. If all the cells of the column j were already queried then the column contains a cell with $(0, \perp, \perp, s)$ by the adversary strategy. So, in either case we are able to form a *leave* l_j in each of the non-marked column.

Now using the cell of *special element* $a_{p,s}$ construct a rooted tree of pointers isomorphic to tree T as defined in Definition 18 such that the internal nodes we will use the other unqueried cells and assign pointers such that $l(j)$'s are the leaves of the tree and the *special element* $a_{p,s}$ is the *root* of the tree. Finally assign anything to the other cell. Now the function will evaluates to 1.

To carry out this construction we need at most $3n$ number of unqueried cells. Outside of the *marked column* total $n-2$ cells for the internal nodes of the tree, atmost $n-1$ unqueried cell for the *leaves* and the $all-1$ *unique marked column* contains total n cell, so total $3n$ unqueried cell will be sufficient for our purpose.

Now there are total n number of columns and to ensure that each of the column in $\{1, 2, \ldots, \frac{n}{2}\}$ contains at least $\frac{n}{2}$ queried cell we need at least $\frac{n^2}{4}$ number of queries. Since $n^2 - 3n \geq \frac{n^2}{4}$ for all $n \geq 6$. Hence $D(\mathsf{A1}_{(n,n)}) = \Omega(n^2)$. ◀

Hence Theorem 20 follows.

Also [17]'s function realises quadratic separation between D and deg and the proof goes via UC_{min} upper bound. But $\mathsf{A1}_{(n,n)}$ exhibits the same properties corresponding to UC_{min}. So, from the following observation it follows that $\mathsf{A1}_{(n,n)}$ also achieves quadratic separation between D and deg.

▶ **Observation 22.** *It is easy to observe that for each positive input x to the function $\mathsf{A1}_{(n,n)}$, the* marked column *together with the rooted tree of pointers with* leaves *in every other column gives a unique minimal 1-certificate of x. Thus, $\mathsf{UC}_1(\mathsf{A1}_{(n,n)}) = \widetilde{O}(n)$. Now, from the definition of UC_{min} it follows that $\mathsf{UC}_{min}(\mathsf{A1}_{(n,n)}) \leq \mathsf{UC}_1(\mathsf{A1}_{(n,n)})$. Hence, $\mathsf{UC}_{min}(\mathsf{A1}_{(n,n)}) = \widetilde{O}(n)$. $\mathsf{UC}_{min}(\mathsf{A1}_{(n,n)}) = \widetilde{O}(n)$. From the fact $\mathsf{UC}_{min} \geq \deg$ it follows that $\deg(\mathsf{A1}_{(n,n)}) = O(n)$.*

▶ **Observation 23** ([5])**.** *For any input $\Sigma^{n \times n}$ to the function $\mathsf{A1}_{(n,n)}$ (in Definition 18) if we permute the rows of the matrix using a permutation σ_r and permute the columns of the matrix using a permutation σ_c and we update the pointers in each of the cells of the matrix accordingly then the function value does not change.*

3 High level description of our techniques

Pointer functions are defined over a special domain called *pointer matrix*, which is a $m \times n$ grid matrix. Each cell of the matrix contains some labels and some pointers that point either to some other cell or to a row or column [6]. For more details, refer to Appendix 2.3. As described in [17], the high level idea of pointer functions is the usage of pointers to make certificates unambiguous without increasing the input size significantly. This technique turns

[6] We naturally think of a pointer pointing to a cell as two pointers - one pointing to the row and the other to the column.

out to be very useful to give separations between various complexity measures as we see in [24], [16] and [5].

Now we want to produce a new function that possesses all the properties of pointer functions, along with the additional property of being transitive. To do so, first, we will encode the labels so that we can permute the bits (by a suitable transitive group) while keeping the structure of unambiguous certificates intact so that the function value remains invariant. One such natural technique would be to encode the contents of each cell in such a way that allows us to permute the bits of the contents of each cell using a transitive group and permute the cells among each other using another transitive group, and doing all of these while ensuring the unambiguous certificates remains intact [7]. This approach has a significant challenge: namely how to encode the pointers.

The information stored in each cell (other than the pointers) can be encoded using fixed logarithmic length strings of different Hamming weights - so that even if the strings are permuted and/or the bits in each string are permuted, the content can be "decoded". Unfortunately, this can only be done when the cell's contents have a constant amount of information - which is the case for pointer functions (except for the pointers). Since the pointers in the cell are strings of size $O(\log n)$ (as they are pointers to other columns or rows), if we want to use the similar Hamming weight trick, the size of the encoding string would need to be polynomial in $O(n)$. That would increase the size of the input compared to the unambiguous certificate. This would not give us tight separation results.

Also, there are three more issues concerning the encodings of pointers:

- As we permute the cells of the matrix according to some transitive group, the pointers within each cell need to be appropriately changed. In other words, when we move some cell's content to some other cell, the pointers pointing to the previous cell should point to the current cell now.
- If a pointer is encoded using a certain t-bit string, different permutations of bits of the encoded pointer can only generate a subset of all t-bit strings.
  For example: if we encode a pointer using a string of Hamming weight 10 then however we permute the bits of the string, the pointer can at most be modified to point to cells (or rows or columns) the encoding of whose pointers also have Hamming weight 10. (The main issue is that permuting the bits of a string cannot change the Hamming weight of a string).
  The encoding of all the pointers should have the same Hamming weight.
- The encoding of the pointers has to be transitive. That is, we should be able to permute the bits of the encodings of the pointer using a transitive group in such a way that either the pointer value does not change or as soon as the pointer values changes, the cells gets permuted accordingly - kind of like an "entanglement".

The above three problems are somewhat connected. Our first innovative idea is to use *binary balance representation* (Definition 3) to represent the pointers. This way, we take care of the second issue. For the first and third issues, we define the transitive group - both the group acting on the contents of the cells (and hence on the encoding of the pointers) and the group acting on the cells itself - in a "entangled" manner. For this we induce a group action

[7] Here, we use the word "encode" since we can view the function defined only over codewords, and when the input is not a codeword, then it evaluates to 0. In our setting, since we are trying to preserve the one-certificates, the codewords are those strings where the unambiguous certificate is encoded correctly. At the same time, we must point out that the encoding of an unambiguous certificate is not necessarily unique.

acting on the nodes of a *balanced binary tree* and generate a transitive subgroup in S_n and $S_{2\log n}$ with the same action which will serve our purpose (Definition 10, Claim 12). This helps us to permute the rows (or columns) using a permutation while updating the encoding of the pointers accordingly.

By Claim 12, for every allowed permutation σ acting on the rows (or columns), there is a unique $\widehat{\sigma}$ acting on the encodings of the pointers in each of the cells such that the pointers are updated according to σ. This still has a delicate problem. Namely, each pointer is either pointing to a row or column. But the permutation $\widehat{\sigma}$ has no way to understand whether the encoding on which it is being applied points to a row or column. To tackle this problem, we think of the set of rows and columns as a single set. All of them are encoded by a string of size (say) $2t$, where for the rows, the second half of the encoding is all 0 while the columns have the first t bits all 0. This is the encoding described in Definition 13 using binary balanced representation. However, this adds another delicate issue about permuting between the first t bits of the encoding and the second t bits.

To tackle this problem, we modify the original function appropriately. We define a slightly modified version of existing pointer functions called $\mathsf{ModA1}$. This finally helps us obtain our "transitive pointer function," which has almost the same complexities as the original pointer function.

We have so far only described the high-level technique to make the 1st variation of pointer functions (Definition 18) transitive where there is the same number of rows and columns. The further variations need more delicate handling of the encoding and the transitive groups - though the central idea is similar.

4 Proof of Theorem 1

4.1 Transitive Pointer Function F_1 for Theorem 1

Our function $F_1 : \Gamma^{n\times n} \to \{0,1\}$ is a composition of two functions - an outer function $\mathsf{ModA1}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ and an inner function $\mathsf{Dec} : \Gamma \to \bar{\Sigma}$. We will set Γ to be $\{0,1\}^{96\log n}$.

The outer function is a modified version of the $\mathsf{A1}_{(n,n)}$ - pointer function described in [5] (see Definition 18 for a description). The function $\mathsf{A1}_{(n,n)}$ takes as input a $(n\times n)$-matrix whose entries are from a set Σ and the function evaluates to 1 if a certain kind of 1-cell-certificate exists. Let us define a slightly modified function $\mathsf{ModA1}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ where $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$. We can think of an input $A \in \bar{\Sigma}^{n\times n}$ as a pair of matrices $B \in \Sigma^{n\times n}$ and $C \in \{\vdash, \dashv\}^{n\times n}$. The function $\mathsf{ModA1}_{(n,n)}$ is defined as

$$\mathsf{ModA1}_{(n,n)}(A) = 1 \text{ iff } \begin{cases} \text{Either, (i)} & \mathsf{A1}_{(n,n)}(B) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \vdash \text{ in the corresponding cells in } C \\ \text{Or, (ii)} & \mathsf{A1}_{(n,n)}(B^T) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \dashv \text{ in the corresponding cells in } C^T \end{cases}$$

Note that both the two conditions (i) and (ii) cannot be satisfied simultaneously. From this it is easy to verify that the function $\mathsf{ModA1}_{(n,n)}$ has all the properties as $\mathsf{A1}_{(n,n)}$ as described in Theorem 19.

The inner function Dec (we call it a decoding function) is function from Γ to $\bar{\Sigma}$, where $\Gamma = 96\log n$. Thus our final function is

$$F_1 := \left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right) : \Gamma^{n\times n} \to \{0,1\}.$$

4.1.1 Inner Function Dec

The input to $\mathsf{A1}_{(n,n)}$ is a Type_1 pointer matrix $\Sigma^{n\times n}$. Each cell of a Type_1 pointer matrix contains a 4-tuple of the form (Value, LPointer, RPointer, BPointer) where Value is either 0 or 1 and LPointer, RPointer are pointers to the other cells of the matrix and BPointer is a pointer to a column of the matrix (or can be a null pointer also). Hence, $\Sigma = \{0,1\} \times [n]^2 \times [n]^2 \times [n]$. For the function $\mathsf{A1}_{(n,n)}$ it was assumed (in [5]) that the elements of Σ is encoded as a k-length[8] binary string in a canonical way.

The main insight for our function $F_1 := \left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ is that we want to maintain the basic structure of the function $\mathsf{A1}_{(n,n)}$ (or rather of $\mathsf{ModA1}_{(n,n)}$) but at the same time we want to encode the $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$ in such a way that the resulting function becomes transitive. To achieve this, instead of having a unique way of encoding an element in $\bar{\Sigma}$ we produce a number of possible encodings[9] for any element in $\bar{\Sigma}$. The inner function Dec is therefore a decoding algorithm that given any proper encoding of an element in $\bar{\Sigma}$ will be able to decode it back.

For the ease of understanding we start by describing the possible "encodings" of $\bar{\Sigma}$, that is by describing the pre-images of any element of $\bar{\Sigma}$ in the function Dec.

"Encodings" of the content of a cell in $\bar{\Sigma}^{n\times n}$:

We will encode any element of $\bar{\Sigma}$ using a string of size $96\log n$ bits. Recall that, an element in $\bar{\Sigma}$ is of the form $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$, where V is the Boolean value, (r_L,c_L), (r_R,c_R) and c_B are the left pointer, right pointers and bottom pointer respectively and T take the value $\vdash$ or $\dashv$. The overall summary of the encoding is as follows:

- **Parts:** We will think of the tuple as 7 objects, namely V, r_L, c_L, r_R, c_R, c_B and T. We will use $16\log n$ bits to encode each of the first 6 objects. The value of T will be encoded in a clever way. So the encoding of any element of $\bar{\Sigma}$ contains 6 *parts* - each a binary string of length $16\log n$.
- **Blocks:** Each of 6 *parts* will be further broken into 4 *blocks* of equal length of $4\log n$. One of the blocks will be a special block called the "encoding block".

Now we explain, for a tuple $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ what is the 4 blocks in each part. We will start by describing a "standard-form" encoding of a tuple $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T = \vdash$. Then we will extend it to describe the standard for encoding of $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T = \dashv$. And finally we will explain all other valid encoding of a tuple $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ by describing all the allowed permutations on the bits of the encoding.

Standard-form encoding of $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T = \vdash$: For the standard-form encoding we will assume that the information of $V, r_L, c_L, r_R, c_R, c_B$ are stored in parts $P1, P2, P3, P4, P5$ and $P6$ respectively. For all $i \in [6]$, the part P_i with have blocks B_1, B_2, B_3 and B_4, of which the block B_1 will be the encoding-block. The encoding will ensure that every parts within a cell will have distinct Hamming weight. The description is also compiled in the Table 2.

- For part $P1$ (that is the encoding of V) the encoding block B_1 will store $\ell_1 \cdot \ell_2$ where ℓ_1 be the $2\log n$ bit binary string with Hamming weight $2\log n$ and ℓ_2 is any $2\log n$ bit binary

[8] For the canonical encoding $k = (1 + 5\log n)$ was sufficient

[9] We use the term "encoding" a bit loosely in this context as technically an encoding means a unique encoding. What we actually mean is the pre-images of the function Dec.

...	B_1 "encoding"-block	B_2	B_3	B_4	Hamming weight
$P1$	$\ell_1\ell_2$, where $\lvert\ell_1\rvert = 2\log n$, and $\lvert\ell_2\rvert = 2\log n - 1 - V$	$4\log n$	$2\log n + 1$	$2\log n + 2$	$12\log n + 2 - V$
$P2$	$\mathcal{E}(r_L, 0)$	$2\log n + 3$	$2\log n + 1$	$2\log n + 2$	$7\log n + 6$
$P3$	$\mathcal{E}(0, c_L)$	$2\log n + 4$	$2\log n + 1$	$2\log n + 2$	$7\log n + 7$
$P4$	$\mathcal{E}(r_R, 0)$	$2\log n + 5$	$2\log n + 1$	$2\log n + 2$	$7\log n + 8$
$P5$	$\mathcal{E}(0, c_R)$	$2\log n + 6$	$2\log n + 1$	$2\log n + 2$	$7\log n + 9$
$P6$	$\mathcal{E}(0, c_B)$	$2\log n + 7$	$2\log n + 1$	$2\log n + 2$	$7\log n + 10$

Table 2 Standard form of encoding of element $(V, (r_L, c_L), (r_R, c_R), c_B, \vdash)$ by a $96\log n$ bit string that is broken into 6 parts $P_1, \ldots, P_6$ of equal size and each Part is further broken into 4 Blocks B_1, B_2, B_3 and B_4. So all total there are 24 blocks each containing a $4\log n$-bit string. For the standard form of encoding of element $(V, (r_L, c_L), (r_R, c_R), c_B, \dashv)$ we encode $(V, (r_L, c_L), (r_R, c_R), c_B, \vdash)$ in the standard form as described in the table and then apply the $\mathsf{Swap}_{\frac{1}{2}}$ on each block. The last column of the table indicates the Hamming weight of each Part.

string with Hamming weight $2\log n - 1 - V$. The blocks B_2, B_3 and B_4 will store a $4\log n$ bit string that has Hamming weight $4\log n, 2\log n + 1$ and $2\log n + 2$ respectively. Any fixed string with the correct Hamming weight will do. We are not fixing any particular string for the blocks B_2, B_3 and B_4 to emphasise the fact that we will be only interested in the Hamming weights of these strings.

- The encoding block $B1$ for parts $P2, P3, P4, P5$ and $P6$ will store the string $\mathcal{E}(r_L, 0)$, $\mathcal{E}(0, c_L)$, $\mathcal{E}(r_R, 0)$, $\mathcal{E}(0, c_r)$ and $\mathcal{E}(0, C_B)$ respectively, where $\mathcal{E}$ is the Balanced-pointer-encoding function (Definition 13). For part P_i (with $2 \le i \le 6$) block B_2, B_3 and B_4 will store any $4\log n$ bit string with Hamming weight $2\log n + 1 + i$, $2\log n + 1$ and $2\log n + 2$ respectively.

Standard form encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \dashv$: For obtaining a standard-form encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \dashv$, first we encode $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \vdash$ using the standard-form encoding. Let $(P1, P2, \ldots, P6)$ be the standard-form encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \vdash$. Now for each of the block apply the $\mathsf{Swap}_{\frac{1}{2}}$ operator.

Valid permutation of the standard form: Now we will give a set of valid permutations to the bits of the encoding of any element of $\bar{\Sigma}$. The set of valid permutations are classified into into 3 categories:

1. Part-permutation: The 6 parts can be permuted using any permutation from S_6
2. Block-permutation: In each of the part, the 4 blocks (say B_1, B_2, B_3, B_4) can be permuted is two ways. (B_1, B_2, B_3, B_4) can be send to one of the following

   (a) Simple Block Swap: (B_3, B_4, B_1, B_2) (b) Block Flip: $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$

The "decoding" function $\mathsf{Dec} : \{0,1\}^{96\log n} \to \bar{\Sigma}$:

- Identify the parts containing the encoding of V, r_L, c_L, r_R, c_R and c_B. This is possible because every part has a unique Hamming weight.
- For each part identify the blocks. This is also possible as in any part all the blocks have distinct Hamming weight. Recall, the valid Block-permutations, namely Simple Block Swap and Block Flip. By seeing the positions of the blocks one can understand if flip was applied and to what and using that one can revert the blocks back to the standard-form (recall Definition 3).

- In the part containing the encoding of V consider the encoding-block. If the block is of the form $\{(\ell_1\ell_2)$ such that $|\ell_1| = 2\log n, |\ell_2| \leq 2\log n - 1\}$ then $T = \{\vdash\}$. If the block is of the form $\{(\ell_2\ell_1)$ such that $|\ell_1| = 2\log n, |\ell_2| \leq 2\log n - 1\}$ then $T = \{\dashv\}$.
- By seeing the encoding block we can decipher the original values and the pointers.
- If the $96\log n$ bit string doesn't have the form of a valid encoding, then decode it as $(0, \perp, \perp, \perp)$.

4.2 Proof of Transitivity of the function

We start with describing the transitive group for which F_1 is transitive.

The Transitive Group: We start with describing a transitive group $\mathcal{T}$ acting on the cells of the matrix A. The matrix has rows $r_1, \ldots, r_n$ and columns $c_1, \ldots, c_n$. And we use the encoding function $\mathcal{E}$ to encode the rows and columns. So the index of the rows and columns are encoded using a $4\log n$ bit string. A permutation from $\mathsf{Bt}_{4\log n}$ (see Definition 10) on the indices of a $4\log n$ bit string will therefore induce a permutation on the set of rows and columns which will give us a permutation on the cells of the matrix. We will now describe the group $\mathcal{T}$ acting on the cells of the matrix by describing the permutation group $\widehat{\mathcal{T}}$ acting on the indices of a $4\log n$ bit string. The group $\widehat{\mathcal{T}}$ will be the group $\mathsf{Bt}_{4\log n}$ acting on the set $[4\log n]$. We will assume that $\log n$ is a power of 2. The group $\mathcal{T}$ with be the resulting group of permutations on the cells of the matrix induced by the group $\widehat{\mathcal{T}}$ acting on the indices on the balanced-pointer-encoding. Note that $\mathcal{T}$ is acting on the domain of $\mathcal{E}$ and $\widehat{\mathcal{T}}$ is acting on the image of $\mathcal{E}$. Also $\widehat{\mathcal{T}}$ is a transitive subgroup of $\mathsf{S}_{4\log n}$ from Claim 11.

▶ **Observation 24.** *For any $1 \leq i \leq 2\log n$ consider the permutation "ith-bit-flip" in $\widehat{\mathcal{T}}$ that applies the transposition $(2i-1, 2i)$ to the indices of the balanced-pointer-encoding. Since the $\mathcal{E}$-encoding of the row $(r_k, 0)$ uses the balanced binary representation of k in the first half and all zero sting in the second half, the jth bit in the binary representation of k is stored in the $2j-1$ and $2j$-th bit in the $\mathcal{E}$-encoding of r_i. So the j-th-bit-flip acts on the sets of rows by swapping all the rows with 1 in the j-th bit of their index with the corresponding rows with 0 in the j-th bit of their index. Also, if $i > \log n$ then there is no effect of the i-th-bit-flip operation on the set of rows. Similarly for the columns.*

Using Observation 24 we have the following claim.

▷ Claim 25. The group $\mathcal{T}$ acting on the cells of of the matrix is a transitive group. That is, for all $1 \leq i_1, j_1, i_2, j_2 \leq n$ there is a permutation $\widehat{\sigma} \in \widehat{\mathcal{T}}$ such that $\widehat{\sigma}[\mathcal{E}(i_1, 0)] = \mathcal{E}(i_2, 0)$ and $\widehat{\sigma}[\mathcal{E}(0, j_1)] = (0, j_2)$. Or in other words, there is a $\sigma \in \mathcal{T}$ acting on the cell of the matrix that would take the cell corresponding to row r_{i_1} and column c_{j_1} to the cell corresponding to row r_{i_2} and column c_{j_2}.

From the Claim 25 we see the group $\mathcal{T}$ acting on the cells of of the matrix is a transitive. But it does not touch the contents within the cells of the matrix. But the input to the function F_1 contains element of $\Gamma = \{0,1\}^{96\log n}$ in each cell. So we now need to extend the group $\mathcal{T}$ to a group G that acts on all the indices of all the bits of the input to the function F_1.

Recall that the input to the function F_1 is a $(n \times n)$-matrix with each cell of matrix containing a binary string of length $96\log n$ which has 6 parts of size $16\log n$ each and each part has 4 blocks of size $4\log n$ each. We classify the generating elements of the group G into 4 categories:

1. Part-permutation: In each of the cells the 6 parts can be permuted using any permutation from S_6
2. Block-permutation: In each of the Parts the 4 blocks can be permuted in the following ways. (B_1, B_2, B_3, B_4) can be send to one of the following
   a. Simple Block Swap: (B_3, B_4, B_1, B_2)
   b. Block Flip (#1): $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$
   c. Block Flip (#2)[10]: $(\mathsf{flip}(B_1), \mathsf{flip}(B_2), B_4, B_3)$
3. Cell-permutation: for any $\sigma \in \mathcal{T}$ the following two action has to be done simultaneously:
   a. (Matrix-update) Permute the cells in the matrix according to the permutation σ. This keeps the contents within each cells untouched - it just changes the location of the cells.
   b. (Pointer-update) For each of blocks in each of the parts in each of the cells permute the indices of the $4 \log n$-bit strings according to σ, that is apply $\widehat{\sigma} \in \widehat{\mathcal{T}}$ corresponding to σ.

We now have the following theorems that would prove that the function F_1 is transitive.

▶ **Theorem 26.** G *is a transitive group and the function* F_1 *is invariant under the action of the* G.

Proof of Theorem 26. To prove that the group G is transitive we show that for any indices $p, q \in [96n^2 \log n]$ there is a permutation $\sigma \in \mathsf{G}$ that would take p to q. Recall that the string $\{0,1\}^{96n^2 \log n}$ is a matrix $\Gamma^{(n \times n)}$ with $\Gamma = \{0,1\}^{96 \log n}$ and every element in Γ is broken into 6 parts and each part being broken into 4 block of size $4 \log n$ each. So we can think of the index p as sitting in k_pth position $(1 \leq k_p \leq 4 \log n)$ in the block B_p of the part P_p in the (r_p, c_p)-th cell of the matrix. Similarly, we can think of q as sitting in k_qth position $(1 \leq k_q \leq 4 \log n)$ in the block B_q of the part P_q in the (r_q, c_q)-th cell of the matrix.

We will give a step by step technique in which permutations from G can be applied to move p to q.

Step 1 **Get the positions in the block correct:** If $k_p \neq k_q$ then take a permutation $\widehat{\sigma}$ from $\widehat{\mathcal{T}}$ that takes k_p to k_q. Since $\widehat{\mathcal{T}}$ is a transitive so such a permutation exists. Apply the cell-permutation $\sigma \in \mathcal{T}$ corresponding to $\widehat{\sigma}$. As a result the index p can be moved to a different cell in the matrix but, by the choice of $\widehat{\sigma}$ its position in the block in which it is will be k_q. Without loss of generality, we assume the the cell location does not change.

Step 2 **Get the cell correct:** Using a cell-permutation that corresponds to a series of "bit-flip" operations change r_p to r_q and c_p to c_q. Since one bit-flip operations basically changes one bit in the binary representation of the index of the row or column such a series of operations can be made.

Since each bit-flip operation is executed by applying the bit-flips in each of the blocks so this might have once again changed the position of the index p in the block. But, even if the position in the block changes it must be a flip operation away. Or in other word, since in the beginning of this step $k_p = k_q$, so if k_q is even (or odd) then after the series bit-flip operations the position of p in the block is either k_q or $(k_q - 1)$ (or $(k_q + 1)$).

Step 3 **Align the Part:** Apply a suitable permutation to ensure that the part P_p moves to part P_q. Note this does not change the cell or the block within the part or the position in the block.

[10] Actually this Block flip can be generated by a combination of Simple Block Swap and Block Flip (#1)

Step 4 **Align the Block:** Using a suitable combination of Simple Block Swap and Block Flip ensures the Block number gets matched, that is B_p goes to B_q. In this case the cell or the Part does not change. But depending on whether the Block Flip operation is applied the position in the block can again change. But, the current position in the block k_p is at most one flip away from k_q.

Step 5 **Apply the final flip:** It might so happen that already we a done after the last step. If not we know that the current position in the block k_p is at most one flip away from k_q. So we apply the suitable Block-flip operation. Thus will not change the cell position, Part number, Block number and the position in the block will match.

Hence we have proved that the group G is transitive. Now we show that the the function F_1 is invariant under the action of G, i.e., for any elementary operations π from the group G and for any input $\Gamma^{(n\times n)}$ the function value does not change even if after the input is acted upon by the permutation π.

Case 1: π is a Part-permutation: It is easy to see that the decoding algorithm Dec is invariant under Part-permutation. This was observed in description of the decoding algorithm Dec in Section 4.1.1. So clearly that the function F_1 is invariant under any Part-permutation.

Case 2: π is a Block-permutation: Here also it is easy to see that the decoding algorithm Dec is invariant under Block-permutation. This was observed in description of the decoding algorithm Dec in Section 4.1.1. Thus F_1 is also invariant under any Block-permutation.

Case 3: π is a Cell-permutation From Observation 23 it is enough to prove that when we permute the cells of the matrix we update the points in the cells accordingly.

Let $\pi \in \mathcal{T}$ be a permutation that permutes only the rows of the matrix. By Claim 14, we see that the contents of the cells will be updated accordingly. Similarly if π only permute the columns of the matrix we will be fine.

Finally, if π swaps the row set and the column set (that is if π makes a transpose of the matrix) then for all i row i is swapped with column i and it is easy to see that $\widehat{\pi}[\mathcal{E}(i,0)] = \mathcal{E}(0,i)$. In that case the encoding block of the value part in a cell also gets swapped. This will thus be encoding the T value as $\dashv$. And so the function value will not be affected as the $T = \dashv$ will ensure that one should apply the π that swaps the row set and the column set to the input before evaluating the function. ◀

4.3 Properties of the Function

▷ Claim 27. Deterministic query complexity of F_1 is $\Omega(n^2)$.

Proof. The function $\mathsf{ModA1}_{(n,n)}$ is a "harder" function than $\mathsf{A1}_{(n,n)}$. So $\mathsf{D}(\mathsf{ModA1}_{(n,n)})$ is at least that of $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Now since, F_1 is $\left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ so clearly the $\mathsf{D}(F_1)$ is at least $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Theorem 20 proves that $\mathsf{D}(\mathsf{A1}_{(n,n)})$ is $\Omega(n^2)$. Hence $\mathsf{D}(F_1) = \Omega(n^2)$. ◀

The following Claim 28 follows from the definition of the function $\mathsf{ModA1}_{(n,n)}$.

▷ Claim 28. The following are some properties of the function $\mathsf{ModA1}_{(n,n)}$

1. $\mathsf{R}_0(\mathsf{ModA1}_{(n,n)}) \leq 2\mathsf{R}_0(\mathsf{A1}_{(n,n)}) + O(n \log n)$
2. $\mathsf{Q}(\mathsf{ModA1}_{(n,n)}) \leq 2\mathsf{Q}(\mathsf{A1}_{(n,n)}) + O(n \log n)$
3. $\deg(\mathsf{ModA1}_{(n,n)}) \leq 2\deg(\mathsf{A1}_{(n,n)}) + O(n \log n)$

Finally, from Theorem 44 we see that the $\mathsf{R}_0(F_1)$, $\mathsf{Q}(F_1)$ and $\deg(F_1)$ are at most $O(\mathsf{R}_0(\mathsf{ModA1}_{(n,n)} \cdot \log n)$, $O(\mathsf{Q}(\mathsf{ModA1}_{(n,n)} \cdot \log n)$ and $O(\deg(\mathsf{ModA1}_{(n,n)} \cdot \log n)$, respectively. So combining this fact with Claim 27, Claim 28 and Theorem 19 (from [5]) we have Theorem 1.

5 Conclusion

As far as we know, this is the first paper that presents a thorough investigation on the separations between various pairs of complexity measures for transitive function. The main technical contribution of this paper is to define transitive versions of pointer functions of [2] that have similar complexity measures as that of the original pointer functions while incurring only a poly-logarithmic blowup in the input size. The current best known 'relationships' and 'separations' between various pairs of measures for transitive functions are summarized in the Table 1.

Unfortunately, a number of cells in the table are not tight. In this context, we would like to point out two important directions:

- For some of these cells, the 'separation' results for transitive functions are weaker than that of the general functions. A natural question is the following: why can't we design a transitive version of the general functions that achieve the same separation? For some cases, like the cheat sheet-based functions, we discuss the difficulties and possible directions in Appendix I. For these cases, the natural question would be to obtain some different collection of functions (maybe not transitive) that achieves similar separations.
- In some of the cells in the Table 1 tight bounds are not known, even for the case of general functions. Can the 'relationships' results in these cases possibly be improved for the case of transitive functions?

In the Table 3 we summarize the results on how low can individual complexity measures go for transitive function. Even with the recent results of Huang [18] and Aaronson et al. [3], there are significant gaps between the 'relationships' and 'separations' in this case.

Finally, we would like to ask the question of how the amount of symmetry affects the relationship between various measures. This is in the lines of the recent work [7]. The study about the different types of symmetries like graph properties, cyclically invariant functions, min-term transitive functions, etc. is not new in this area, but a more elaborate analysis is required to quantify the relationship between the measures and the amount of symmetry.

References

1. Scott Aaronson. Quantum certificate complexity. *Journal of Computer and System Sciences*, 74(3):313–322, 2008. doi:10.1016/j.jcss.2007.06.020.
2. Scott Aaronson, Shalev Ben-David, and Robin Kothari. Separations in query complexity using cheat sheets. In *STOC*, pages 863–876, 2016. doi:10.1145/2897518.2897644.
3. Scott Aaronson, Shalev Ben-David, Robin Kothari, Shravas Rao, and Avishay Tal. Degree vs. approximate degree and quantum implications of Huang's sensitivity theorem. In *STOC*, pages 1330–1342, 2021. doi:10.1145/3406325.3451047.
4. Andris Ambainis. Superlinear advantage for exact quantum algorithms. *SIAM Journal on Computing*, pages 617–631, 2016. doi:10.1137/130939043.
5. Andris Ambainis, Kaspars Balodis, Aleksandrs Belovs, Troy Lee, Miklos Santha, and Juris Smotrovs. Separations in query complexity based on pointer functions. *Journal of the ACM*, 64(5):32:1–32:24, 2017. doi:10.1145/3106234.
6. Aleksandrs Belovs and Robert Spalek. Adversary lower bound for the k-sum problem. In *ITCS*, pages 323–328, 2013. doi:10.1145/2422436.2422474.
7. Shalev Ben-David, Andrew M. Childs, András Gilyén, William Kretschmer, Supartha Podder, and Daochen Wang. Symmetries, graph properties, and quantum speedups. In *FOCS*, pages 649–660, 2020. doi:10.1109/FOCS46700.2020.00066.

8 Shalev Ben-David, Pooya Hatami, and Avishay Tal. Low-sensitivity functions from unambiguous certificates. In *ITCS*, volume 67, pages 28:1–28:23, 2017. doi:10.4230/LIPIcs.ITCS.2017.28.

9 Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997. doi:10.1137/S0097539796300933.

10 Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

11 Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002. doi:10.1016/S0304-3975(01)00144-X.

12 Sourav Chakraborty. On the sensitivity of cyclically-invariant Boolean functions. *Discrete Mathematics and Theoretical Computer Science*, 13(4):51–60, 2011. doi:10.46298/dmtcs.552.

13 Andrew Drucker. Block sensitivity of minterm-transitive functions. *Theoretical Computer Science*, 412(41):5796–5801, 2011. doi:10.1016/j.tcs.2011.06.025.

14 Yihan Gao, Jieming Mao, Xiaoming Sun, and Song Zuo. On the sensitivity complexity of bipartite graph properties. *Theoretical Computer Science*, 468:83–91, 2013. doi:10.1016/j.tcs.2012.11.006.

15 Justin Gilmer, Michael E. Saks, and Srikanth Srinivasan. Composition limits and separating examples for some Boolean function complexity measures. *Combinatorica*, 36(3):265–311, 2016. doi:10.1007/s00493-014-3189-x.

16 Mika Göös, T. S. Jayram, Toniann Pitassi, and Thomas Watson. Randomized communication versus partition number. *ACM Transactions on Computation Theory*, 10(1):4:1–4:20, 2018. doi:10.1145/3170711.

17 Mika Göös, Toniann Pitassi, and Thomas Watson. Deterministic communication vs. partition number. *SIAM Journal on Computing*, 47(6):2435–2450, 2018. doi:10.1137/16M1059369.

18 Hao Huang. Induced subgraphs of hypercubes and a proof of the sensitivity conjecture. *Annals of Mathematics*, 190(3):949–955, 2019. doi:10.4007/annals.2019.190.3.6.

19 Shelby Kimmel. Quantum adversary (upper) bound. *Chicago Journal of Theoritical Computer Science*, 2013, 2013. doi:10.4086/cjtcs.2013.004.

20 Troy Lee, Rajat Mittal, Ben W. Reichardt, Robert Spalek, and Mario Szegedy. Quantum query complexity of state conversion. In *FOCS*, pages 344–353, 2011. doi:10.1109/FOCS.2011.75.

21 Troy Lee and Jérémie Roland. A strong direct product theorem for quantum query complexity. *Computational Complexity*, 22(2):429–462, 2013. doi:10.1109/CCC.2012.17.

22 Qian Li and Xiaoming Sun. On the sensitivity complexity of k-uniform hypergraph properties. In *STACS*, volume 66, pages 51:1–51:12, 2017. doi:10.4230/LIPIcs.STACS.2017.51.

23 Ashley Montanaro. A composition theorem for decision tree complexity. *Chicago Journal of Theoritical Computer Science*, 2014, 2014. doi:10.4086/cjtcs.2014.006.

24 Sagnik Mukhopadhyay and Swagato Sanyal. Towards better separation between deterministic and randomized query complexity. In *FSTTCS*, volume 45, pages 206–220, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.206.

25 Noam Nisan and Mario Szegedy. On the degree of Boolean functions as real polynomials. *Computational Complexity*, 4:301–313, 1994. doi:10.1007/BF01263419.

26 Noam Nisan and Avi Wigderson. On rank vs. communication complexity. *Combinatorica*, 15(4):557–565, 1995. doi:10.1007/BF01192527.

27 Ben Reichardt. Reflections for quantum query algorithms. In *SODA*, pages 560–569. SIAM, 2011. doi:10.1137/1.9781611973082.44.

28 David Rubinstein. Sensitivity vs. block sensitivity of Boolean functions. *Combinatorica*, 15(2):297–299, 1995. doi:10.1007/BF01200762.

29 Marc Snir. Lower bounds on probabilistic linear decision trees. *Theoretical Computer Science*, 38:69–82, 1985. doi:10.1016/0304-3975(85)90210-5.

730 **30** Xiaoming Sun. Block sensitivity of weakly symmetric functions. *Theoretical Computer Science*,
731 384(1):87–91, 2007. doi:10.1016/j.tcs.2007.05.020.
732 **31** Xiaoming Sun. An improved lower bound on the sensitivity complexity of graph properties.
733 *Theoretical Computer Science*, 412(29):3524–3529, 2011. doi:10.1016/j.tcs.2011.02.042.
734 **32** Xiaoming Sun, Andrew Chi-Chih Yao, and Shengyu Zhang. Graph properties and circular
735 functions: How low can quantum query complexity go? In *CCC*, pages 286–293, 2004.
736 doi:10.1109/CCC.2004.1313851.
737 **33** Avishay Tal. Properties and applications of Boolean function composition. In *ITCS*, pages
738 441–454, 2013. doi:10.1145/2422436.2422485.
739 **34** György Turán. The critical complexity of graph properties. *Information Processing Letters*,
740 18(3):151–153, 1984. doi:10.1016/0020-0190(84)90019-X.

Measure	known lower bounds	Known example
D	$\Omega(\sqrt{N})$ [32]	$O(\sqrt{N})$ [32]
R_0	$\Omega(\sqrt{N})$	$O(\sqrt{N})$ [32]
R	$\Omega(N^{\frac{1}{3}})$ $\mathsf{bs(f)} = O(\mathsf{R}(f))$	$O(\sqrt{N})$ [32]
C	$\Omega(\sqrt{N})$	$O(\sqrt{N})$ $Tribe(\sqrt{N}, \sqrt{N})$
RC	$\Omega(N^{\frac{1}{3}})$ $\mathsf{bs(f)} = O(\mathsf{RC}(f))$	$O(\sqrt{N})$ $Tribe(\sqrt{N}, \sqrt{N})$
bs	$\Omega(N^{\frac{1}{3}})$ [30]	$\widetilde{O}(N^{\frac{3}{7}})$ [30], [13]
s	$\Omega(N^{\frac{1}{10}})$ $\mathsf{C(f)} = O(\mathsf{s(f)})^5$	$\Theta(N^{\frac{1}{3}})$ [12]
λ	$\Omega(N^{\frac{1}{12}})$ $\mathsf{C(f)} = O(\lambda(\mathsf{f}))^6$	$\Theta(N^{\frac{1}{3}})$ [12]
$\mathsf{Q_E}$	$\Omega(N^{\frac{1}{4}})$ $\mathsf{Q}(f) = O(\mathsf{Q_E}(f))$	$O(\sqrt{N})$ [32]
$\deg$	$\Omega(N^{\frac{1}{6}})$ $\mathsf{D}(f) = O(\deg(f)^3)$	$O(\sqrt{N})$ [32]
Q	$\Omega(N^{\frac{1}{4}})$ [32]	$\widetilde{O}(N^{\frac{1}{4}})$ [32]
$\widetilde{\deg}$	$\Omega(N^{\frac{1}{8}})$ $\mathsf{D}(f) = O(\widetilde{\deg}(f)^4)$	$\widetilde{O}(N^{\frac{1}{4}})$ [32]

Table 3 In each row, for the measure A, the two entries a, b represents: (1) (Known lower bound) for all transitive Boolean function f , $A(f) = \Omega(a)$, and (2) (Known example) there exists a *transitive* function g such that $A(g) = O(b)$, where a and b are some polynomial in N.

A Known lower bounds for complexity measures for the class of transitive function

Table 3 represents the individual known separations and the known example for different complexity measures for the class of transitive function:

B Complexity measures of Boolean functions

We refer the reader to the survey [11] for an introduction to the complexity of Boolean functions and complexity measures. Several additional complexity measures and their relations among each other can also be found in [8] and [3]. Similar to the above references, we define several complexity measures of Boolean functions that are relevant to us.

We refer to [11] and [5] for definitions of deterministic query model, randomized query model and quantum query model for Boolean functions.

▶ **Definition 29** (Deterministic query complexity). *The deterministic query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{D}(f)$*, is the worst-case cost of the best deterministic query algorithm computing* f*.*

▶ **Definition 30** (Randomized query complexity). *The randomized query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{R}(f)$*, is the worst-case cost of the best randomized query algorithm computing* f *to error at most* $1/3$*.*

▶ **Definition 31** (Zero-error randomized query complexity). *The zero-error randomized query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{R}_0(f)$*, is the minimum worst-case expected cost of a randomized query algorithm that computes* f *to zero-error, that is, on every input* x *the algorithm should give the correct answer* $f(x)$ *with probability* 1*.*

▶ **Definition 32** (Quantum query complexity). *The quantum query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{Q}(f)$*, is the worst-case cost of the best quantum query algorithm computing* f *to error at most* $1/3$*.*

▶ **Definition 33** (Exact quantum query complexity). *The exact quantum query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{Q}_\mathsf{E}(f)$*, is the minimum number of queries made by a quantum algorithm that outputs* $f(x)$ *on every input* $x \in \{0,1\}^n$ *with probability* 1*.*

Next, we define the notion of partial assignment that will be used to define several other complexity measures.

▶ **Definition 34** (Partial assignment). *A partial assignment is a function* $p : S \to \{0,1\}$ *where* $S \subseteq [n]$ *and the size of* p *is* $|S|$*. For* $x \in \{0,1\}^n$ *we say* $p \subseteq x$ *if* x *is an extension of* p*, that is the restriction of* x *to* S *denoted* $x|_S = p$*.*

▶ **Definition 35** (Certificate complexity). *A* 1*-certificate is a partial assignment which forces the value of the function to* 1 *and similarly a* 0*-certificate is a partial assignment which forces the value of the function to* 0*. The certificate complexity of a function* f *on* x*, denoted as* $\mathsf{C}(x, f)$*, is the size of the smallest* $f(x)$*-certificate that can be extended to* x*.*

Also, define 0*-certificate of* f *as* $\mathsf{C}^0(f) = \max\{\mathsf{C}(f, x) : x \in \{0,1\}^n, f(x) = 0\}$ *and* 1*-certificate of* f *as* $\mathsf{C}^1(f) = \max\{\mathsf{C}(f, x) : x \in \{0,1\}^n, f(x) = 1\}$*. Finally, define the certificate complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{C}(f)$*, to be* $\max\{\mathsf{C}^0(f), \mathsf{C}^1(f)\}$*.*

▶ **Definition 36** (Unambiguous certificate complexity). *For any Boolean function* $f : \{0,1\}^n \to \{0,1\}$*, a set of partial assignments* U *is said to form an unambiguous collection of* 0*-certificates for* f *if*

1. *Each partial assignment in* U *is a* 0*-certificate (with respect to* f*)*
2. *For each* $x \in f^{-1}(0)$*, there is some* $p \in U$ *with* $p \subseteq x$
3. *No two partial assignments in* U *are consistent.*

We then define $\mathsf{UC}_0(f)$ *to be the minimum value of* $\max_{p \in U} |p|$ *over all choices of such collections* U*. We define* $\mathsf{UC}_1(f)$ *analogously, and set* $\mathsf{UC}(f) = \max\{\mathsf{UC}_0(f), \mathsf{UC}_1(f)\}$*. We also define the one-sided version,* $\mathsf{UC}_{\min}(f) = \min\{\mathsf{UC}_0(f), \mathsf{UC}_1(f)\}$*.*

Next we define randomized certificate complexity (see [1]).

▶ **Definition 37** (Randomized certificate complexity). *A randomized verifier for input* x *is a randomized algorithm that, on input* y *in the domain of* f *accepts with probability* 1 *if* $y = x$*, and rejects with probability at least* $\frac{1}{2}$ *if* $f(y) \neq f(x)$*. If* $y \neq x$ *but* $f(y) = f(x)$*, the acceptance probability can be arbitrary. The Randomized certificate complexity of* f *on input* x *is denoted by* $\mathsf{RC}(f, x)$*, is the minimum expected number of queries used by a randomized verifier for* x*. The randomized certificate complexity of* f*, denoted by* $\mathsf{RC}(f)$*, is defined as* $\max\{\mathsf{RC}(f, x) : x \in \{0,1\}^n\}$*.*

▶ **Definition 38** (Block sensitivity). *The block sensitivity* $\mathsf{bs}(f,x)$ *of a function* $f:\{0,1\}^n\to\{0,1\}$ *on an input* x *is the maximum number of disjoint subsets* $B_1,B_2,\ldots,B_r$ *of* $[n]$ *such that for all* j, $f(x)\neq f(x^{B_j})$, *where* $x^{B_j}\in\{0,1\}^n$ *is the input obtained by flipping the bits of* x *in the coordinates in* B_j. *The block sensitivity of* f, *denoted by* $\mathsf{bs}(f)$, *is* $\max\{\mathsf{bs}\{(f,x)\}:x\in\{0,1\}^n\}$.

▶ **Definition 39** (Sensitivity). *The sensitivity of* f *on an input* x *is defined as the number of bits on which the function is sensitive:* $\mathsf{s}(f,x)=|\{i:f(x)\neq f(x^i)\}|$. *We define the sensitivity of* f *as* $\mathsf{s}(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n\}$

We also define 0*-sensitivity of* f *as* $\mathsf{s}^0(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n,f(x)=0\}$, *and* 1*-sensitivity of* f *as* $\mathsf{s}^1(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n,f(x)=1\}$.

Here we are defining *Spectral sensitivity* from [3]. For more details about *Spectral sensitivity* and it's updated relationship with other complexity measures we refer [3].

▶ **Definition 40** (Spectral sensitivity). *Let* $f:\{0,1\}^n\to\{0,1\}$ *be a Boolean function. The* sensitivity graph *of* f, $G_f=(V,E)$ *is a subgraph of the Boolean hypercube, where* $V=\{0,1\}^n$, *and* $E=\{(x,x\oplus e_i)\in V\times V:i\in[n],f(x)\neq f(x\oplus e_i)\}$, *where* $x\oplus e_i\in V$ *is obtained by flipping the* i*th bit of* x. *That is,* E *is the set of edges between neighbors on the hypercube that have different* f*-value. Let* A_f *be the adjacency matrix of the graph* G_f. *We define the spectral sensitivity of* f *as the largest eigenvalue of* A_f.

▶ **Definition 41** (Degree). *A polynomial* $p:\mathbb{R}^n\to\mathbb{R}$ *represents* $f:\{0,1\}^n\to\{0,1\}$ *if for all* $x\in\{0,1\}^n$, $p(x)=f(x)$. *The degree of a Boolean function* f, *denoted by* $\mathsf{deg}(f)$, *is the degree of unique multilinear polynomial that represents* f.

▶ **Definition 42** (Approximate degree). *A polynomial* $p:\mathbb{R}^n\to\mathbb{R}$ *approximately represents a function* $f:\{0,1\}^n\to\{0,1\}$ *if for all* $x\in\{0,1\}^n$, $|p(x)-f(x)|\leq\frac{1}{3}$. *The approximate degree of a Boolean function* f, *denoted by* $\widetilde{\mathsf{deg}}(f)$, *is the minimum degree of a polynomial that approximately represents* f.

The following is a known relation between degree and one-sided unambiguous certificate complexity measure ([8]).

▶ **Observation 43** ([8]). *For any Boolean function* f, $\mathsf{UC}_{\mathsf{min}}(f)\geq\mathsf{deg}(f)$.

One often try to understand how the complexity measure of composed function compare with respect to the measures of the individual functions. The following folklore theorem that we will be using multiple times in out paper.

▶ **Theorem 44.** *Let* $f:\{0,1\}^n\to\{0,1\}$ *and* $g:\{0,1\}^m\to\{0,1\}^k$ *be two Boolean functions then*

1. $\mathsf{D}(f\circ g)=\Omega(\mathsf{D}(f)/k)$, *assuming* g *is an onto function.*
2. $\mathsf{Q}(f\circ g)=\Omega(\mathsf{Q}(f)/k)$, *assuming* g *is onto.*
3. $\mathsf{R}_0(f\circ g)=O(\mathsf{R}_0(f)\cdot m)$ *and if* g *is onto then* $\mathsf{R}_0(f\circ g)=O(\mathsf{R}_0(f)/m)$
4. $\mathsf{R}(f\circ g)=O(\mathsf{R}(f)\cdot m)$ *and if* g *is onto then* $\mathsf{R}(f\circ g)=O(\mathsf{R}(f)/m)$.
5. $\widetilde{\mathsf{deg}}(f\circ g)=O(\widetilde{\mathsf{deg}}(f)\cdot m)$
6. $\widetilde{\mathsf{deg}}(f\circ g)=O(\widetilde{\mathsf{deg}}(f)\cdot m)$.

If the inner function g is Boolean valued then we can obtain some tighter results for the composed functions.

▶ **Theorem 45.** *Let* $f:\{0,1\}^n\to\{0,1\}$ *and* $g:\{0,1\}^m\to\{0,1\}$ *be two Boolean functions then*

1. *([33, 23])* $\mathsf{D}(f \circ g) = \Theta(\mathsf{D}(f) \cdot \mathsf{D}(g))$.
2. *([27, 20, 19])* $\mathsf{Q}(f \circ g) = \Theta(\mathsf{Q}(f) \cdot \mathsf{Q}(g))$.
3. *(folklore)* $\deg(f \circ g) = \Theta(\deg(f) \cdot \deg(g))$.

Finally the following observation proves that composition of transitive functions is also a transitive function.

▶ **Observation 46.** *Let $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^m \to \{0,1\}$ be transitive functions. Then $f \circ g : \{0,1\}^{nm} \to \{0,1\}$ is also transitive.*

Proof. Let $T_f \subseteq \mathsf{S}_n$ and $T_g \subseteq \mathsf{S}_m$ be the transitive groups corresponding to f and g, respectively. On input $x = (X_1, \ldots, X_n)$, $X_i \in \{0,1\}^m$ for $i \in [n]$, the function $f \circ g$ is invariant under the action of the group $T_f \wr T_g$ - the wreath product of the T_f with T_g. The group $T_f \wr T_g$ acts on the input string through the following permutations:

1. any permutation $\pi \in T_f$ acting on indices $\{1, \ldots, n\}$ or
2. any permutations $(\sigma_1, \ldots, \sigma_n) \in (T_g)^n$ acting on $X_1, \ldots, X_n$ i.e. $(\sigma_1, \ldots, \sigma_n)$ sends $X_1, \ldots, X_n$ to $\sigma_1(X_1), \ldots, \sigma_n(X_n)$.

◀

Over the years a number of interesting Boolean functions has been constructed to demonstrate differences between various measures of Boolean functions. Some of the functions has been referred to in the Table 1. We describe the various functions in the Appendix C.

B.1 Proof of Claim 11 and Claim 12

We restate the claims here for convenience.

▷ Claim 47 (Restatement of Claim 11). The group Bt_k is a transitive group.

Proof of Claim 11. For any $i, j \in [k]$, we have to show that there exists a permutation $\pi \in \mathsf{Bt}_k$ such that $\pi(i) = j$. Let us form a complete binary tree of height $\log k$ in the following way:

- (Base case:) Start from root node, label the left and right child as 0 and 1 respectively.
- For every node x, label the left and right child as $x0$ and $x1$ respectively.

Note that our complete binary tree has k leaves, where each of the leaf is labeled by a binary string of the form $x_1 x_2 \ldots x_{\log k}$, which is the binary representation of numbers in $[k]$. Similarly any node in the tree can be labeled by a binary string $x_1 x_2 \ldots x_t$, where $0 \leq t \leq \log k$ and t is the distance of the node from the root.

Now for any $i, j \in [k]$, let the binary representation of i be $(x_1 x_2 \ldots x_{\log k})$ and that of j be $(y_1 y_2 \ldots y_{\log k})$. Now we will construct the permutation $\pi \in \mathsf{Bt}_k$ such that $\pi(i) = j$. Without loss of generality, we can assume $i \neq j$.

Find the least positive integer $\ell \in [\log k]$ such that $x_\ell \neq y_\ell$, then go to the node labeled $x_1 x_2 \ldots x_{\ell-1}$ and swap it's left and right child. Let $\pi_{x_1 \ldots x_{\ell-1}} \in S_k$ be the corresponding permutation of the leaves of the tree, in other words on the set $[k]$. Note that, by definition, the permutation $\pi_{x_1 \ldots x_{\ell-1}} \in S_k$ is in Bt_k. Also note that the permutation $\pi_{x_1 \ldots x_{\ell-1}}$ acts of the set $[k]$ as follows:

- $\pi_{x_1 \ldots x_{\ell-1}}(z_1 \ldots z_{\log k}) = z_1 \ldots z_{\log k}$ if $z_1 \ldots z_{\ell-1} \neq x_1 \ldots x_{\ell-1}$
- $\pi_{x_1 \ldots x_{\ell-1}}(x_1 \ldots x_{\ell-1} 0 z_{\ell+1} \ldots z_{\log k}) = (x_1 \ldots x_{\ell-1} 1 z_{\ell+1} \ldots z_{\log k})$
- $\pi_{x_1 \ldots x_{\ell-1}}(x_1 \ldots x_{\ell-1} 1 z_{\ell+1} \ldots z_{\log k}) = (x_1 \ldots x_{\ell-1} 0 z_{\ell+1} \ldots z_{\log k})$

Since $i = x_1 \dots x_{\ell-1} x_\ell \dots x_{\log k}$ and $j = y_1 \dots y_{\ell-1} y_\ell \dots y_{\log k}$ with $x_1 \dots x_{\ell-1} = y_1 \dots y_{\ell-1}$ and $x_\ell \neq y_\ell$, so

$$\pi_{x_1 \dots x_{\ell-1}}(i) = y_1 \dots y_{\ell-1} y_\ell x_{\ell+1} \dots y_{\log k}$$

So the binary representation of $\pi_{x_1 \dots x_{\ell-1}}(i)$ and j matching in the first ℓ positions which is one more that the number of positions where the binary representation of i and j matched. By doing this trick repeatedly, that is by applying different permutations from Bt_k one after another we can map i to j. ◀

▷ Claim 48 (Restatement of Claim 12). For all $\widehat{\gamma} \in \mathsf{Bt}_{2\log n}$ there is a $\gamma \in \mathsf{S}_n$ such that for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

Proof of Claim 12. Recall the group $\mathsf{Bt}_{2\log n}$: assuming that the elements of $[2\log n]$ are placed on the leaves of the binary tree of depth $\log(2\log n)$, the group $\mathsf{Bt}_{2\log n}$ is generated by the permutations of the form "pick a node in the binary tree of and swap the left and right sub-tree of the node". So it is enough to prove that for any elementary permutation $\widehat{\gamma}$ of the form "pick a node in the binary tree and swap the left and right sub-tree of the node" there is a $\gamma \in \mathsf{S}_n$ such that for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

Any node in the binary tree of depth $\log(2\log n)$ can be labeled by a 0/1-string of length t, where $0 \leq t \leq \log(2\log n)$ is the distance of the node from the root. We split the proof of the claim into two cases depending on the value of the t - the distance from the root.

B.1.0.1 If $t = \log(2\log n)$:

This is the case when the node is at the last level - just above the leaf level. Let the node be u and let s be the number of whose binary representation is the label of the node u. Let the numbers in the leaves of the tree corresponds to the $bb(i)$ - the balanced binary representation of $i \in [n]$. Note that because of the balanced binary representation the children of u are

- 0 (left-child) and 1 (right-child) if the s-th bit in the binary representation of i is 0
- 1 (left-child) and 0 (right-child) if the s-th bit in the binary representation of i is 1

So the permutation (corresponding to swapping the left and right sub-trees of u) only change the order of 0 and 1 - which corresponds to flipping the s-th bit of the binary representation of i. And so in this case the γ acting on the set $[n]$ is just collection transpositions swapping i and j iff the the binary representation of i and j are same except for the s-th bit.

So in this case for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

B.1.0.2 If $t < \log(2\log n)$:

Let the node be v. Note that in this case since the node keeps the order of the $2r-1$ and $2r$ bits unchanged (for any $1 \leq r \leq \log n$), so it is enough we can visualise the action by an action of swapping the left and right sub-trees of the node v on the binary representation of i (instead of the balance binary representation of i). And so we can see that the action of the permutation (corresponding to swapping the left and right sub-trees of v) automatically gives a permutation of the binary representations of numbers between 1 and n, as was discussed in the proof of Claim 11. And hence we have for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

◀

C Some Boolean functions and their properties

In this section we define some standard functions that are either mentioned in the Table 1 or used somewhere in the paper. We also state some of their properties that we need for our proofs. We start by defining some basic Boolean functions.

▶ **Definition 49.** *Define* PARITY $: \{0,1\}^n \to \{0,1\}$ *to be the* PARITY$(x_1,\ldots,x_n) = \sum x_i mod\ 2$. *We use the notation* $\oplus$ *to denote* PARITY.

▶ **Definition 50.** *Define* AND $: \{0,1\}^n \to \{0,1\}$ *to be the* AND$(x_1,\ldots,x_n) = 0$ *if and only if there exists an* $i \in [n]$ *such that* $x_i = 0$. *We use the notation* $\wedge$ *to denote* AND.

▶ **Definition 51.** *Define* OR $: \{0,1\}^n \to \{0,1\}$ *to be the* OR$(x_1,\ldots,x_n) = 1$ *if and only if there exists an* $i \in [n]$ *such that* $x_i = 1$. *We use the notation* $\vee$ *to denote* OR.

▶ **Definition 52.** *Define* MAJORITY $: \{0,1\}^n \to \{0,1\}$ *as* MAJORITY$(x) = 1$ *if and only if* $|x| > \frac{n}{2}$.

We need the following definition of composing iteratively with itself.

▶ **Definition 53** (Iterative composition of a function)**.** *Let* $f : \{0,1\}^n \to \{0,1\}$ *a Boolean function. For* $d \in \mathbb{N}$ *we define the function* $f^d : \{0,1\}^{n^d} \to \{0,1\}$ *as follows: if* $d = 1$ *then* $f^d = f$, *otherwise*

$$f^d(x_1,\ldots,x_{n^d}) = f\left(f^{d-1}(x_1,\ldots,x_{n^{d-1}}),\ldots,f^{d-1}(x_{n^d-n+1},\ldots,x_{n^d})\right).$$

▶ **Definition 54.** *For* $d \in \mathbb{N}$ *define* NAND*-tree of depth* d *as* NAND^d *where* NAND $: \{0,1\}^2 \to \{0,1\}$ *is defined as:* NAND$(x_1,x_2) = 0$ *if and only if* $x_1 \neq x_2$. *We use the notation* $\widetilde{\wedge}$*-tree to denote* NAND*-tree.*

The now define a function that gives a quadratic separation between sensitivity and block sensitivity.

▶ **Definition 55** (Rubinstein's function ([28]))**.** *Let* $g : \{0,1\}^k \to \{0,1\}$ *be such that* $g(x) = 1$ *iff* x *contains two consecutive ones and the rest of the bits are* 0. *The Rubinstein's function, denoted by* RUB $: \{0,1\}^{k^2} \to \{0,1\}$ *is defined to be* RUB $=$ OR$_k \circ g$.

▶ **Theorem 56.** *[28] For the Rubinstein's function in Definition 55* s(RUB) $= k$ *and* bs(RUB) $= k^2/2$. *Thus* RUB *witnesses a quadratic gap between sensitivity and block sensitivity.*

[25] first introduced a function whose deg is significantly smaller than s or bs. This appears in the footnote in [26] that E. Kushilevitz also introduced a similar function with 6 variables which gives slightly better gap between s and deg. Later Ambainis computed $\mathsf{Q_E}$ of that function and gave a separation between $\mathsf{Q_E}$ and D [4]. This function is fully sensitive at all zero input, consequently this gives a separation between $\mathsf{Q_E}$ and s.

▶ **Definition 57** ([25])**.** *Define* NW *as follows:*

$$\mathsf{NW}(x_1,x_2,x_3) = \begin{cases} 1 \text{ iff } x_i \neq x_j \text{ for some } i,j \in \{1,2,3\} \\ 0 \text{ otherwise.} \end{cases}$$

Now define the d*-th iteration* NW^d *on* $(x_1,x_2,\ldots,x_{3^d})$ *as Definition 53 where* $d \in \mathbb{N}$.

▶ **Definition 58** (Kushilevitz's function). *Define* K *as follows:*

$$\mathsf{K}(x_1,x_2,x_3,x_4,x_5,x_6) = \Sigma x_i + \Sigma x_i y_i + x_1x_3x_4 + x_1x_4x_5 + x_1x_2x_5 + x_2x_3x_4 + x_2x_3x_5 + x_1x_2x_6 + x_1x_3x_6 + x_2x_4x_6 + x_3x_5x_6 + x_4x_5x_6.$$

Now define the d-th iteration K^d *on* $(x_1, x_2, \ldots, x_{6^d})$ *as Definition 53 where* $d \in \mathbb{N}$.

Next we will describe two example that was introduced in [15] and gives the separation between C vs. RC and RC vs. bs respectively . They also have introduces some new complexity measures for iterative version of a function and how to use them to get the critical measure between two complexity measures. For more details we refer to [15].

▶ **Definition 59** ([15]). *Let* n *be an even perfect square, let* $k = 2\sqrt{n}$ *and* $d = \sqrt{n}$. *Divide the* n *indices of the input into* n/k *disjoint blocks. Define* $\mathsf{GSS}_1 : \{0,1\}^n \to \{0,1\}$ *as follows:* $\mathsf{GSS}_1(x) = 1$ *if and only if* $|x| \geq d$ *and all the* 1*'s in* x *are in a single group. Define* GSS_1^d *with* $\mathsf{GSS}_1^1 = f$.

▶ **Definition 60** ([15]). *Define* $\mathsf{GSS}_2 : \{0,1\}^n \to \{0,1\}$ *where* n *is of the form* $\binom{t}{2}$ *for some integer* t. *Identify the input bits of* GSS_2 *with the edges of the complete graph* K_t. *An input* $x \in \{0,1\}^n$ *induces a subgraph of* K_t *consisting of edges assigned* 1 *by* x. *The function* $\mathsf{GSS}_2(x)$ *is defined to be* 1 *iff the subgraph induces by* x *has a star graph.*

▶ **Definition 61.** *For* $\Sigma = [n^k]$ *the function* $\mathsf{k\text{-}sum} : \Sigma^n \to \{0,1\}$ *is defined as follows: on input* $x_1, x_2, \ldots, x_n \in \Sigma$, *if there exists* k *element* $x_{i_1}, \ldots, x_{i_k}$, $i_1, \ldots, i_k \in [n]$, *that sums to* 0 $(mod\ |\Sigma|)$ *then output* 1, *otherwise output* 0.

▶ **Theorem 62** ([2, 6]). *For the function* $\mathsf{k\text{-}sum} : \Sigma^n \to \{0,1\}$, *if* $|\Sigma| \geq 2\binom{n}{n}$ *then*

$$\mathsf{Q}(\mathsf{k\text{-}sum}) = \Omega(n^{k/(k+1)}/\sqrt{k}).$$

Next, we define the cheat sheet version of a function from [2].

▶ **Definition 63.** *We define cheat sheet version of* f *as follows: the input to* f_{CS} *consist of* $\log n$ *inputs to* f , *each of size* n , *followed by* n *blocks of bits of size* $C(f) \times \log n$ *each. Let us denote the input to* f_{CS} *as* $X = (x_1, x_2, \ldots, x_{\log n}, Y_1, Y_2, ..., Y_n)$, *where* x_i *is an input to* f , *and the* Y_i *are the aforementioned cells of size* $C(f) \times \log n$. *The first part* $x_1, x_2, \ldots, x_{\log n}$ *of the string, we call them as input section and the rest of the part we call as certificate section of the whole input. Define* $f_{CS} : \{0,1\}^{n \times \log n + n \times (C(f) \log n} \to \{0,1\}$ *to be* 1 *if and only if the following conditions hold:*

- *For all* i, x_i *is in the domain of* f . *If this condition is satisfied, let* l *be the positive integer corresponding to the binary string* $(f(x_1), f(x_2), ..., f(x_{\log n}))$.
- Y_l *certifies that all* x_i *are in the domain of* f *and that* l *equals the binary string formed by their output values,* $(f(x_1), f(x_2), ..., f(x_{\log n}))$.

Finally, we present the pointer functions and its variants introduced in [5]. They are used for demonstrate separation between several complexity complexity measures like *deterministic query complexity*, *Randomized query complexity*, *Quantum query complexity* etc. These functions was originally motivated from [17] function. The functions that we construct for many of our theorems is a composition function whose outer function is these pointer function, or a slight variant of these. In the next section we present the formal definition of pointer functions.

C.1 Variants of Pointer function

Now we describe *Variant 2* of [5] function where the domain is slightly different from that of *Variant 1*.

▶ **Definition 64** (Variant 2 [5]). *Let* $\Sigma^{m\times n}$ *be a* Type_2 pointer matrix *such that BPointer* $\in \{(i,j)|i\in[m], j\in[n]\}$. *Here the BPointer points to a cell of* M, *not a column. Let* n *be even. Define* $\mathsf{A2}_{(m,n)} : \Sigma^{m\times n} \to \{0,1\}$ *on* Type_2 pointer matrix *such that for all* $x = (x_{i,j}) \in \Sigma^{m\times n}$, *the function* $\mathsf{A2}_{(m,n)}(x_{i,j})$ *evaluates to* 1 *if and only if it has a* 1-cell certificate *of the following form:*

1. *there exists exactly one* Marked column $j^\star$ *in* M. *Let* $(i^\star, j^\star)$ *be the* special element.
2. *for each non-marked column* $j \in [n]\setminus\{b\}$ *there exist an element* l_j *such that* $Value(l_j) = 0$ *where* l_j *be the end of the path that starts at the* special element *and follows the pointers LPointer and RPointer as specified by the sequence* $T(j)$. l_j *exists in all* $j \in [n]\setminus\{b\}$. *We call that* l_j *the Leaves of the tree.*
3. *The size of the set* $\{j \in [n]\setminus\{b\} | BPointer(l_j) = (i^\star, j^\star)\}$ *is exactly* $\frac{n}{2}$.

Variant 2 achieves the separation between $\mathsf{R_0}$ vs. R and $\mathsf{R_0}$ vs $\mathsf{Q_E}$. The following results are some of the properties of *Variant 2* function:

▶ **Theorem 65.** *[5] For any* $m, n \in \mathbb{N}$, *the function* $\mathsf{A2}_{(m,n)}$ *in Definition 64 satisfies*

$$\mathsf{R_0} = \Omega(mn),$$
$$\mathsf{R} = \widetilde{O}(m+n) \text{ and}$$
$$\mathsf{Q_E} = \widetilde{O}(m+n).$$

Finally we will describe *Variant 3* of [5] function, where there is one extra parameter, the number of *marked column*. So, we will address the function as $\mathsf{A3}_{(k,m,n)}$ where m, n are number of rows and columns respectively and k is the number of *marked column*.

▶ **Definition 66** (Variant 3 [5]). *Define* $\mathsf{A3}_{(k,m,n)} : \Sigma^{m\times n} \to \{0,1\}$ *on* Type_2 pointer-matrix Σ *such that for all* $x = (x_{i,j}) \in \Sigma^{m\times n}$, *the function* $\mathsf{A3}_{(k,m,n)}(x_{i,j})$ *evaluates to* 1 *if and only if it has a* 1-cell certificate *of the following form:*

1. *there exists* k Marked column *in* M. *Let* $b_1, b_2, \ldots, b_k \in [n]$ *denotes the* Marked column *in* M *and* (a_k, b_k) *be the* special element *in each* Marked column. *Let us denote the cell of* special element *by a*
2. $BPointer(x_{a_j,b_j}) = (a_{j+1}, b_{j+1})$ *for all* $j \in [k-1]$ *and* $BPointer(x_{a_k,b_k}) = (a_1, b_1)$. *Also,* $LPointer(x_{a_s,b_s}) = LPointer(x_{a_t,b_t})$ *and* $RPointer(x_{a_s,b_s}) = RPointer(x_{a_t,b_t})$ *for all* $s, t \in [k]$.
3. *for each non-marked column* $j \in [n]\setminus\{b_1, b_2, \ldots, b_k\}$ *there exist an element* ℓ_j *such that* $Value(l_j) = 0$ *and* $BPointer(l_j) = b$ *where* l_j *be the end of the path that starts at the* special element *and follows the pointers LPointer and RPointer as specified by the sequence* $T(j)$. *We need that* l_j *exists in each* $j \in [n]\setminus\{b_1, b_2, \ldots, b_k\}$ *i.e. no pointer on the path is* $\perp$. *We call that* l_j *the Leaves of the tree.*

Variant 3 realises the separation between $\mathsf{R_0}$ vs. Q, R vs. $\mathsf{Q_E}$, R vs. deg and $\widetilde{\deg}$ for different m, n and k. The followings are some important result for *variant 3* from [5]:

▶ **Theorem 67.** *[5] For sufficiently large $m, n \mathbb{N}$ and a natural number $k < n$, the function $\mathsf{A3}_{(k,m,n)}$ in Definition 66 satisfies*

$$\mathsf{R}_0 = \Omega(mn) \text{ for } k < \frac{n}{2},$$
$$\mathsf{R}_0 = \Omega(\frac{mn}{\log n}) \text{ for } k = 1,$$
$$\mathsf{Q} = \widetilde{O}(\sqrt{mn/k} + \sqrt{km} + k + \sqrt{n}),$$
$$\mathsf{Q}_\mathsf{E} = \widetilde{O}(m\sqrt{n/k} + km + n) \text{ and}$$
$$\widetilde{\deg} = \widetilde{O}(\sqrt{m} + \sqrt{n}) \text{ for } k = 1.$$

▶ **Observation 68.** *It is easy to observe that for each positive input x to the function $\mathsf{A3}_{(1,n,n)}$, the* marked column *together with the rooted tree of pointers with* leaves *in every other column gives a unique minimal 1-certificate of x. Thus, $\mathsf{UC}_1(\mathsf{A3}_{(1,n,n)}) = \widetilde{O}(n)$. Now, from the definition of UC_{min} it follows that $\mathsf{UC}_{min}(\mathsf{A3}_{(n,n)}) = \widetilde{O}(n)$. Also it follows from the fact $\mathsf{UC}_{min} \geq \deg$ that $\deg(\mathsf{A3}_{(n,n)}) = O(n)$.*

D Separation between zero-error randomized query complexity and some of other complexity measures

[2] used the a variant of the pointer function to give separation between zero-error randomized query complexity(R_0) and other measures like one-sided randomized query complexity, exact quantum query complexity and degree. In this section we construct a transitive version of the functions that was used in [2] to show such separations and we prove the following theorem:

▶ **Theorem 69.** *There exists a transitive function F_{69} such that*

$$\mathsf{R}_0(F_{69}) = \widetilde{\Omega}(\mathsf{R}(F_{69})^2), \qquad \mathsf{R}_0(F_{69}) = \widetilde{\Omega}(\mathsf{Q}_\mathsf{E}(F_{69})^2), \qquad \mathsf{R}_0(F_{69}) = \widetilde{\Omega}(\deg(F_{69})^2).$$

D.1 The Function

Recall from Definition 15, the input to the function $\mathsf{A2}$ is a Type_2 *pointer matrix* $\Sigma^{n\times n}$. Each cell of such matrix is of the form $(\text{Value}, \text{LPointer}, \text{RPointer}, \text{BPointer})$ where Value is either 0 or 1 and LPointer, RPointer and BPointer are pointers to the other cells of the matrix (or can be a null pointer also). Here the BPointer are the pointers to the cells, which is different from $\mathsf{A1}_{(n,n)}$. That's why we need an encoding which is slightly different from the encoding scheme of $\mathsf{A1}_{(n,n)}$. Our function $F_{69} : \Gamma^{n\times n} \to \{0,1\}$ is again a composition of two functions - an outer function $\mathsf{ModA2}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ and an inner function $\mathsf{Dec}_1 : \Gamma \to \bar{\Sigma}$ where Γ is $\{0,1\}^{112\log n}$. Here the function Dec_1 has different domain from the previous one described in Section 4.1.1.

First we define the outer function $\mathsf{ModA2}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ where $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$ to be the modified version of the $\mathsf{A2}_{(n,n)}$ as we have defined $\mathsf{ModA1}_{(n,n)}$.

Think of an input $A \in \bar{\Sigma}^{n\times n}$ as a pair of matrices $B \in \Sigma^{n\times n}$ and $C \in \{\vdash, \dashv\}^{n\times n}$. The function $\mathsf{ModA2}_{(n,n)}$ is defined as

$$\mathsf{ModA2}_{(n,n)}(A) = 1 \text{ iff } \begin{cases} \text{Either, (i)} & \mathsf{A2}_{(n,n)}(B) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \vdash \text{ in the corresponding cell in } C \\ \text{Or, (ii)} & \mathsf{A2}_{(n,n)}(B^T) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate has } \dashv \text{ in the corresponding cell in } C^T \end{cases}$$

...	B_1 "encoding"-block	B_2	B_3	B_4	Hamming weight
$P1$	$\ell_1\ell_2$, where $\lvert\ell_1\rvert = 2\log n$, and $\lvert\ell_2\rvert = 2\log n - 1 - V$	$4\log n$	$2\log n + 1$	$2\log n + 2$	$12\log n + 2 - V$
$P2$	$\mathcal{E}(r_L, 0)$	$2\log n + 3$	$2\log n + 1$	$2\log n + 2$	$7\log n + 6$
$P3$	$\mathcal{E}(0, c_L)$	$2\log n + 4$	$2\log n + 1$	$2\log n + 2$	$7\log n + 7$
$P4$	$\mathcal{E}(r_R, 0)$	$2\log n + 5$	$2\log n + 1$	$2\log n + 2$	$7\log n + 8$
$P5$	$\mathcal{E}(0, c_R)$	$2\log n + 6$	$2\log n + 1$	$2\log n + 2$	$7\log n + 9$
$P6$	$\mathcal{E}(r_B, 0)$	$2\log n + 7$	$2\log n + 1$	$2\log n + 2$	$7\log n + 10$
$P7$	$\mathcal{E}(0, c_B)$	$2\log n + 8$	$2\log n + 1$	$2\log n + 2$	$7\log n + 11$

Table 4 Standard form of encoding of element $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), \vdash)$ by a $112\log n$ bit string that is broken into 7 parts $P1, \ldots, P7$ of equal size and each Part is further broken into 4 Blocks B_1, B_2, B_3 and B_4. So all total there are 24 blocks each containing a $4\log n$-bit string. For the standard form of encoding of element $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), \dashv)$ we encode $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), \vdash)$ in the standard form as described in the table and then apply the $\mathsf{Swap}_{\frac{1}{2}}$ on each block. The last column of the table indicates the Hamming weight of each Part.

Consequently the function $\mathsf{ModA2}_{(n,n)}$ has all the properties as $\mathsf{A2}_{(n,n)}$ as described in Theorem 65.

The inner function Dec_1 (we call it a decoding function) is function from Γ to $\bar{\Sigma}$, where $\Gamma = 112\log n$. Thus our final function is

$$F_{69} := \left(\mathsf{ModA2}_{(n,n)} \circ \mathsf{Dec}_1\right) : \Gamma^{n\times n} \to \{0,1\}.$$

D.1.1 Inner Decoding Function

In the similar fashion of Section 4.1.1 we start by describing the standard "encodings" of a single element of $\bar{\Sigma}$, then we will describe a multiple possible ways of encoding a single element of the $\bar{\Sigma}$ and it's decoding scheme.

"Encodings" of the content of a cell in $\bar{\Sigma}^{n\times n}$ where $\Sigma^{n\times n}$ is a Type_2 *Pointer matrix*

We will encode any element of $\bar{\Sigma}$ using a string of size $112\log n$ bits. Recall from Definition 15 that in case of Type_2 *pointer matrix* an element in $\bar{\Sigma}$ is of the form $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$, where V is a binary value and $T \in \{\vdash, \dashv\}$. The overall summary of the encoding is compiled in the Table 4:

- **Parts:** We will think of the tuple as 8 objects, namely V, r_L, c_L, r_R, c_R, r_B, c_B and T. We will use $16\log n$ bits to encode each of the first 7 objects. So the encoding of any element of $\bar{\Sigma}$ contains 7 *parts*-each a binary string of length $16\log n$. The value of T will be encoded in a kind-of hidden way.
- **Blocks:** Each of 7 parts will be further broken into 4 blocks of equal length of $4\log n$. One of the block will be a special block called the "encoding block".

Now We will start by describing a "standard-form" encoding of a tuple $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \vdash$. Then we will extend it to describe the standard for encoding of $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \dashv$. And finally we will explain all other valid encoding of a tuple $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ by describing all the allowed operations on the bits of the encoding.

Standard form encoding of $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \vdash$

For the standard-form encoding we will assume that the information of $V, r_L, c_L, r_R, c_R, r_B, c_B$ are stored in parts $P1, P2, P3, P4, P5$, $P6$ and $P7$ respectively. For all $i \in [7]$, the part P_i with have blocks B_1, B_2, B_3 and B_4, of which the block B_1 will be the encoding-block. The description of the standard-form encoding is also compiled in the Table 4.

Standard form encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ **where** $T = \dashv$

For obtaining a standard-form encoding of $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \dashv$, first we encode $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \vdash$ using the standard-form encoding in Table 4. Let $(P1, P2, \ldots, P7)$ be the standard-form encoding of $(V, (r_L, c_L), (r_R, c_R), (r_B, c_B), T)$ where $T = \vdash$. Now for each of the block apply the following $\mathsf{Swap}_{\frac{1}{2}}$ operator.

Valid permutation of the standard form

Now we will give a set of valid permutations to the bits of the encoding of any element of $\bar{\Sigma}$. The set of valid permutations are classified into into 3 categories:

1. Part-permutation: The 7 parts can be permuted using any permutation from S_7
2. Block-permutation: In each of the part, the 4 blocks (say B_1, B_2, B_3, B_4) can be permuted is two ways. (B_1, B_2, B_3, B_4) can be send to one of the following
   a. Simple Block Swap: (B_3, B_4, B_1, B_2)
   b. Block Flip: $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$

We are now ready to describe the decoding function Dec_1 from $\Gamma = \{0,1\}^{112 \log n}$ to $\bar{\Sigma}$.

- Identify the parts containing the encoding of V, r_L, c_L, r_R, c_R, r_B and c_B. This is possible because every part has a unique Hamming weight.
- For each part identify the blocks. This is also possible as in any part all the blocks have distinct Hamming weight. By seeing the positions of the blocks one can understand if flip was applied and to what and using that one case revert the blocks back to the standard-form.
- In the part containing the encoding of V consider the encoding-block. If the block is of the form $\{(\ell_1 \ell_2)\ such\ that\ |\ell_1| = 2 \log n, |\ell_2| \leq 2 \log n - 1\}$ then $T = \{\vdash\}$. If the block is of the form $\{(\ell_2 \ell_1)\ such\ that\ |\ell_1| = 2 \log n, |\ell_2| \leq 2 \log n - 1\}$ then $T = \{\dashv\}$.
- By seeing the encoding block we can decipher the original values and the pointers.
- If the $112 \log n$ bit string doesn't have the form of a valid encoding, then decode it as $(0, \perp, \perp, \perp)$.

D.2 Proof of transitivity of the function

▶ **Theorem 70.** *F_{69} is transitive under the action of the transitive group G_1.*

The group G_1 is formed with same permutation sets as G except S_6 that was acting on the *Part*. Instead of S_6 we will take S_7 as Part-permutation. Proof of Theorem 70 follows from similar argument of Theorem 26. Here everything else is similar as function F_1 except *Part*.

D.2.1 Properties of the function

▷ Claim 71. Zero error randomized query complexity of F_{69} is $\Omega(n^2)$.

Proof. The function $\mathsf{ModA2}_{(n,n)}$ is a "harder" function than $\mathsf{A2}_{(n,n)}$. So $\mathsf{R}_0(\mathsf{ModA2}_{(n,n)})$ is at least that of $\mathsf{R}_0(\mathsf{A2}_{(n,n)})$. Now since, F_{69} is $\left(\mathsf{ModA2}_{(n,n)} \circ \mathsf{Dec}_1\right)$ so clearly the $\mathsf{R}_0(F_{69})$ is at least $(\mathsf{R}_0(\mathsf{A2}_{(n,n)})/\log n)$ by Theorem 44. From Theorem 65 we have $\mathsf{R}_0(F_{69})$ at least $\Omega(n^2)$. ◀

The following Claim 72 that follows from the definition of the function $\mathsf{ModA2}_{(n\times n)}$.

▷ Claim 72. The following are some properties of the function $\mathsf{ModA2}_{(n,n)}$

1. $\mathsf{R}(\mathsf{ModA2}_{(n,n)}) \leq 2\mathsf{R}(\mathsf{A2}_{(n,n)}) + O(n \log n)$
2. $\mathsf{Q_E}(\mathsf{ModA2}_{(n,n)}) \leq 2\mathsf{Q_E}(\mathsf{A2}_{(n,n)}) + O(n \log n)$
3. $\deg(\mathsf{ModA2}_{(n,n)}) \leq 2\deg(\mathsf{A2}_{(n,n)}) + O(n \log n)$

Finally, from Theorem 44 we see that $\mathsf{R_0}(F_{69})$, $\mathsf{Q_E}(F_{69})$ and $\deg(F_{69})$ are at most $O(\mathsf{R_0}(\mathsf{ModA2}_{(n\times n)} \cdot \log n)$, $O(\mathsf{Q_E}(\mathsf{ModA2}_{(n\times n)} \cdot \log n)$ and $O(\deg(\mathsf{ModA2}_{(n\times n)} \cdot \log n)$, respectively. So combining this fact with Claim 71, Claim 72 and Theorem 65 (from [5]) we have Theorem 69.

E Separation between randomized query complexity and some of other complexity measures

[2] considered yet another variant of the pointer function to show gap between randomized query complexity(R) and complexity measures like approximate degree and degree. In this section we show how obtain a transitive version of this function, thus proving the following theorem:

▶ **Theorem 73.** *There exists a transitive function F_{73} such that*

$$\mathsf{R}(F_{73}) = \widetilde{\Omega}(\widetilde{\deg}(F_{73})^4), \qquad \mathsf{R}(F_{73}) = \widetilde{\Omega}(\deg(F_{73})^2).$$

Using standard techniques we can obtain the following theorems as corollaries to Theorem 73.

▶ **Theorem 74.** *There exists a transitive function F_{74} such that $\mathsf{R_0}(F_{74}) = \widetilde{\Omega}(\mathsf{Q}(F_{74})^3)$.*

▶ **Theorem 75.** *There exists a transitive function F_{75} such that $\mathsf{R}(F_{75}) = \widetilde{\Omega}(\mathsf{Q_E}(F_{75})^{1.5})$.*

E.1 Transitive function for Theorem 73

In [5] the function $\mathsf{A3}_{(1,n,n)}$ achieves the separation between R and $\widetilde{\deg}$. Before that [16] introduced a function that achieves quadratic separation between R and deg. The function $\mathsf{A3}$ is motivated from [16] function and generates the same separation between R and deg. We will give a transitive function that is motivated from [5] and achieves the separation between R vs. deg and R vs. $\widetilde{\deg}$ which is same as general function.

First let us define the outer function $\mathsf{ModA3}_{(1,n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ where $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$ to be the modified version of the $\mathsf{A3}_{(1,n,n)}$ as we have defined $\mathsf{ModA1}_{(1,n,n)}$.

Think of an input $A \in \bar{\Sigma}^{n\times n}$ as a pair of matrices $B \in \Sigma^{n\times n}$ and $C \in \{\vdash, \dashv\}^{n\times n}$. The function $\mathsf{ModA3}_{(1,n,n)}$ is defined as

$$\mathsf{ModA3}_{(1,n,n)}(A) = 1 \text{ iff } \begin{cases} \text{Either, (i)} & \mathsf{A3}_{(1,n,n)}(B) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \vdash \text{ in the corresponding cell in } C \\ \text{Or, (ii)} & \mathsf{A3}_{(1,n,n)}(B^T) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate has } \dashv \text{ in the corresponding cell in } C^T \end{cases}$$

Consequently the function $\mathsf{ModA3}_{(1,n,n)}$ has all the properties of $\mathsf{A3}_{(1,n,n)}$ as described in Theorem 67. Here the input matrix to the function $\mathsf{A3}_{(1,n,n)}$ is a Type_2 pointer matrix which is same as $\mathsf{A2}_{(n,n)}$. Hence we can encode the elements in $\bar{\Sigma}$ in a similar fashion that

we have done in Section D.1.1. That's why we can choose the same decoding function Dec_1 function as of F_{69}. Thus our final function is

$$F_{73} := \left(\mathsf{ModA3}_{(1,n,n)} \circ \mathsf{Dec}_1\right) : \Gamma^{n\times n} \to \{0,1\}$$

where $\mathsf{Dec}_1 : \Gamma \to \bar{\Sigma}$ and $\Gamma = 112 \log n$.

From the proof of Theorem 70 we have the following Theorem:

▶ **Theorem 76.** *F_{73} is transitive under the action of the transitive group G_1.*

▷ Claim 77. Randomized query complexity of F_{73} is $\Omega(n^2)$.

Proof. From the construction of $\mathsf{ModA3}_{(1,n,n)}$ one can note that $\mathsf{ModA3}_{(1,n,n)}$ is a "harder" function than $\mathsf{A3}_{(1,n,n)}$. So $\mathsf{R}(\mathsf{ModA3}_{(1,n,n)})$ is at least that of $\mathsf{R}(\mathsf{A3}_{(1,n,n)})$.

Now since, F_{73} is $\left(\mathsf{ModA3}_{(1,n,n)} \circ \mathsf{Dec}_1\right)$ from Theorem 44 it follows that $\mathsf{R}(F_{73})$ is at least $\mathsf{R}(\mathsf{A3}_{(1,n,n)})$. In [5] they proved that $\mathsf{R}(\mathsf{A3}_{(1,n,n)})$ is $\Omega(n^2)$. So $\mathsf{R}(F_{73})$ is at least $\Omega(\frac{n^2}{\log n})$. ◀

▷ Claim 78. The following are some properties of the function $\mathsf{ModA3}_{(1,n,n)}$

1. $\widetilde{\deg}(\mathsf{ModA3}_{(1,n,n)}) \leq 2\widetilde{\deg}(\mathsf{A3}_{(1,n,n)}) + O(n \log n)$
2. $\deg(\mathsf{ModA3}_{(1,n,n)}) \leq 2\deg(\mathsf{A3}_{(1,n,n)}) + O(n \log n)$

Finally from Theorem 44, Theorem 67 and Claim 78 it follows that $\widetilde{\deg}(F_{73}) = O(\sqrt{n})$. Also from Theorem 44, Observation 68 and Claim 78 it follows that $\deg(F_{73}) = (n)$. From Theorem 76 and Claim 77 we have Theorem 73.

E.2 Transitive function for Theorem 74

In [5] they proved that $\mathsf{A3}_{(n,n,n^2)}$ achieves the separation between R_0 and Q where the input is a $n \times n^2$ Type_2 *pointer matrix*. Till now all of our encoding was precisely for $n \times n$ matrix. Those encoding schemes directly cannot be generalized for any non-square matrix. But using some modifications to our encoding scheme in the previous sections, we give an encoding for $n \times n^2$ matrix also. Before going into the encoding scheme we will define some useful terms and representation of a matrix that we are going to use in the encoding scheme.

▶ **Definition 79** (*Brick*). *For any $n \times n^2$ matrix M we divide the matrix into total n number of square matrices each of size $n \times n$, which we refer to as* Brick. *Thus there is total n number of* Bricks, *which we denote as $\{b_1, b_2, \ldots, b_n\}$ and each b_i is a sub-matrix of M. Now let us denote the n rows and n columns of an $n \times n$ matrix as $\{r_1, \ldots, r_n\}$ and $\{c_1, \ldots, c_n\}$ respectively. So, each entries of M can be uniquely defines as $\{(r, c, b) | r, c, b \in [n]\}$ where r, c and b is the row number, column number and brick number respectively. In this set-up we will write an $n \times n^2$ matrix A as $A_{(n\times n\times n)}$.*

In a general setup when we swap between row and column we get the transpose of that matrix. In our row, column and brick setup we define two new concept similar to transpose matrix.

▶ **Definition 80** ($A^\top, A^\dashv$). *For any $n \times n^2$ matrix $A_{(n\times n\times n)}$ if each entry is denoted by $(row, column, brick)$ then we define $A^\top_{(n\times n\times n)}$ to be the matrix where (r, c, b)-th element of $A_{(n\times n\times n)}$ is the (b, r, c)-th element of $A^\top_{(n\times n\times n)}$. Similarly we define $A^\dashv_{(n\times n\times n)}$ to be the matrix where (r, c, b)-th element of $A_{(n\times n\times n)}$ is the (c, b, r)-th element of $A^\dashv_{(n\times n\times n)}$.*

Now we are ready to define our outer function.

E.2.1 The outer function

The outer function is a modified version of the $\mathsf{A3}_{(n,n,n^2)}$. Recall, from Definition 66, the function $\mathsf{A3}_{(n,n,n^2)}$ takes as input a $(n \times n^2)$ Type_2 *pointer matrix* over Σ.

Let us define $\mathsf{ModA3}*_{(n,n,n^2)} : \bar{\Sigma}^{n\times n^2} \to \{0,1\}$ where the alphabet set is $\bar{\Sigma} = \Sigma \times \{\vdash, \top, \dashv\}$. We can think of an input $M \in \bar{\Sigma}^{n\times n^2}$ as a pair of matrices $B \in \Sigma^{n\times n^2}$ and $C \in \{\vdash, \top, \dashv\}^{n\times n^2}$. The function $\mathsf{ModA3}*_{(n,n,n^2)}$ is defined as

$$\mathsf{ModA3}*_{(n,n,n^2)}(M) = 1 \text{ iff } \begin{cases} \text{Either,} & \textbf{(i)}\ \mathsf{A3}_{(n,n,n^2)}(B) = 1, \text{ and, all the cells in the 1-cell-certificate have } \vdash \text{ in the corresponding cell in } C \\ \text{Or,} & \textbf{(ii)}\ \mathsf{A3}_{(n,n,n^2)}(B^{\top}) = 1, \text{ and, all the cells in the 1-cell-certificate has } \top \text{ in the corresponding cell in } C^{\top} \\ \text{Or,} & \textbf{(iii)}\ \mathsf{A3}_{(n,n,n^2)}(B^{\dashv}) = 1, \text{ and, all the cells in the 1-cell-certificate has } \dashv \text{ in the corresponding cell in } C^{\dashv} \end{cases}$$

At most one of conditions (i), (ii) and (iii) can be true for an input. From this it is easy to verify that the function $\mathsf{ModA3}*_{(n,n,n^2)}$ has all the properties as $\mathsf{A3}_{(n,n,n^2)}$ as described in Theorem 67.

Thus our final function is

$$F_{74} := \left(\mathsf{ModA3}*_{(n,n,n^2)} \circ \mathsf{Dec}_2\right) : \Gamma^{n^3} \to \{0,1\},$$

where the inner function Dec_2 (we call it a decoding function) is a function from Γ to $\bar{\Sigma}$ where $\Gamma = 240 \log n$.

E.2.2 Inner Decoding Function

Any element of Type_2 pointer matrix is of the form $(V, \text{LPointer}, \text{RPointer}, \text{BPointer})$ where $V \in \{0,1\}$ and LPointer, RPointer and BPointer are the pointers to the other cell of the matrix. Here pointers are of the form (i,j) where i is the row number and j is the column number. Now in our setup (Definition 79) we have represented the cells of an $n \times n^2$ matrix as (r,c,b) (where $r,c,b \in [n]$), where r, c and b specify the row, column and brick of that cell, respectively.

"Encodings" of the content of a cell in $\bar{\Sigma}^{n\times n^2}$ where $\Sigma^{n\times n^2}$ is a Type_2 *Pointer matrix*

We will encode any tuple of the form $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ which is an element of $\bar{\Sigma}$ using a string of size $240 \log n$ bits. The overall summary of the encoding is as follows:

- **Parts:** We will think of the tuple as 11 objects, namely V, r_L, c_L, b_L r_R, c_R, b_R, r_B, c_B, b_B and T. We will use a $24 \log n$ bit string to encode each of the first 10 objects and the enoding of T will be implicit. So the encoding of any element of $\bar{\Sigma}$ contains 10 *parts*-each a binary string of length $24 \log n$.
- **Blocks:** Each of the 10 parts will be further broken into 4 blocks of equal length of $6 \log n$. One of the block will be a special block called the "encoding block".

We will start by describing a "standard-form" encoding of a tuple $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ where $T = \vdash$. Then extend it for $T = \top$ and $T = \dashv$. And finally we will explain all other valid encoding of a tuple $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ by describing all the allowed operations on the bits of the encoding.

...	B_1	B_2	B_3	B_4	Hamming weight
$P1$	$\ell_1\ell_3\ell_2$ where $\lvert\ell_1\rvert = 2\log n$ $\lvert\ell_2\rvert = 2\log n - 1$, and $\lvert\ell_3\rvert = 2\log n - 2 - V$	$6\log n$	$2\log n + 1$	$2\log n + 2$	$16\log n - V$
$P2$	$\mathcal{E}'(i)$	$2\log n + 3$	$2\log n + 1$	$2\log n + 2$	$7\log n + 6$
$P3$	$\mathcal{E}'(j)$	$2\log n + 4$	$2\log n + 1$	$2\log n + 2$	$7\log n + 7$
$P4$	$\mathcal{E}'(k)$	$2\log n + 5$	$2\log n + 1$	$2\log n + 2$	$7\log n + 8$
$P5$	$\mathcal{E}'(t)$	$2\log n + 6$	$2\log n + 1$	$2\log n + 2$	$7\log n + 9$
$P6$	$\mathcal{E}'(m)$	$2\log n + 7$	$2\log n + 1$	$2\log n + 2$	$7\log n + 10$
$P7$	$\mathcal{E}'(n)$	$2\log n + 8$	$2\log n + 1$	$2\log n + 2$	$7\log n + 11$
$P8$	$\mathcal{E}'(p)$	$2\log n + 9$	$2\log n + 1$	$2\log n + 2$	$7\log n + 12$
$P9$	$\mathcal{E}'(q)$	$2\log n + 10$	$2\log n + 1$	$2\log n + 2$	$7\log n + 13$
$P10$	$\mathcal{E}'(r)$	$2\log n + 11$	$2\log n + 1$	$2\log n + 2$	$7\log n + 14$

Table 5 Standard form of encoding of element $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),\vdash)$ by a $240\log n$ bit string that is broken into 10 parts $P1,\ldots,P10$ of equal size and each Part is further broken into 4 Blocks B_1,B_2,B_3 and B_4. So all total there are 40 blocks each containing a $6\log n$-bit string. For the standard form of encoding of element $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),\top)$ we encode $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),\vdash)$ in the standard form as described in the table and then apply the $\mathsf{Rotation}_1$ on each block. For the standard form of encoding of element $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),\dashv)$ we encode $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),\vdash)$ in the standard form as described in the table and then apply the $\mathsf{Rotation}_2$ on each block.The last column of the table indicates the unique Hamming weight of each Part.

Using the Balanced-Binary Encoding we can define an hybrid encoding for the set of rows and columns.

▶ **Definition 81.** *Given a set R of n rows $r_1,\ldots,r_n$, a set C of n columns $c_1,\ldots,c_n$ and a set B of n* bricks *$b_1,\ldots,b_n$ we define the balanced-pointer-encoding function*

$$\mathcal{E}' : (R\times\{0\}\times\{0\})\cup(\{0\}\times C\times\{0\})\cup(\{0\}\times\{0\}\times B)\to\{0,1\}^{6\log n}$$

as follows:

$$\begin{aligned}\mathcal{E}'(r_i,0,0) &= bb(i)\cdot 0^{2\log n}\cdot 0^{2\log n},\\ \mathcal{E}'(0,c_i,0) &= 0^{2\log n}\cdot bb(i)\cdot 0^{2\log n}\ \textit{and}\\ \mathcal{E}'(0,0,b_i) &= 0^{2\log n}\cdot 0^{2\log n}\cdot bb(i).\end{aligned}$$

▶ **Definition 82.** *Given a binary string $x_1,\ldots,x_{3k}\in\{0,1\}^{3k}$, the operation* $\mathsf{Rotation}_1$ *and* $\mathsf{Rotation}_2$ *act on the string by*

$$\mathsf{Rotation}_1(x_1,\ldots,x_{3k}) = x_{2k+1},\ldots,x_{3k},x_1\ldots,x_k,x_{k+1},\ldots,x_{2k}$$

and

$$\mathsf{Rotation}_2(x_1,\ldots,x_{3k}) = x_{k+1},\ldots,x_{2k},x_{2k+1}\ldots,x_{3k},x_1,\ldots,x_k$$

Now we are set to describe the standard-form encoding of $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),T)$ where $T=\vdash$.

Standard form encoding of $(V,(r_L,c_L,b_L),(r_R,c_R,b_R),(r_B,c_B,b_B),T)$ **where** $T=\vdash$

For the standard-form encoding we will assume that the information of $V, r_L, c_L, b_L, r_R, c_R, b_R, r_B, c_B, b_B$ are stored in parts $P1, P2, P3, P4, P5$, $P6$, $P7$, $P8$, $P9$ and $P10$ respectively. For all $i \in [10]$, the part P_i with have blocks B_1, B_2, B_3 and B_4, of which the block B_1 will be the encoding-block. The description of the standard-form encoding is also compiled in the Table 5.

- For *part* $P1$ (that is the encoding of V) the encoding block B_1 will store $\ell_1 \cdot \ell_2 \cdot \ell_3$ where ℓ_1 is any $2\log n$ bit binary string with Hamming weight $2\log n$, ℓ_2 is any $2\log n$ bit binary string with Hamming weight $2\log n - 1$ and ℓ_3 is any $2\log n$ bit binary string with Hamming weight $2\log n - 2 - V$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $6\log n$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P2$ (this is the encoding of r_L) the encoding block $B1$ will store the string $\mathcal{E}'(r_L, 0, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 3$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P3$ (this is the encoding of c_L) the encoding block $B1$ will store the string $\mathcal{E}'(0, c_L, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 4$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P4$ (this is the encoding of b_L) the encoding block $B1$ will store the string $\mathcal{E}'(0, 0, b_L)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 5$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P5$ (this is the encoding of r_R) the encoding block $B1$ will store the string $\mathcal{E}'(r_R, 0, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 6$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P6$ (this is the encoding of c_R) the encoding block $B1$ will store the string $\mathcal{E}'(0, c_R, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 7$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P7$ (this is the encoding of b_R) the encoding block $B1$ will store the string $\mathcal{E}'(0, 0, b_R)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 8$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P8$ (this is the encoding of r_B) the encoding block $B1$ will store the string $\mathcal{E}'(r_B, 0, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 9$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P9$ (this is the encoding of c_B) the encoding block $B1$ will store the string $\mathcal{E}'(0, c_B, 0)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 10$, $2\log n + 1$ and $2\log n + 2$ respectively.
- For *part* $P10$ (this is the encoding of b_B) the encoding block $B1$ will store the string $\mathcal{E}'(0, 0, b_B)$. Block B_2, B_3 and B_4 will store any $6\log n$ bit string with Hamming weight $2\log n + 11$, $2\log n + 1$ and $2\log n + 2$ respectively.

Standard form encoding of $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ where $T = \top$ and $T = \dashv$

For obtaining a standard-form encoding of $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ where $T = \top$, first we encode $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ where $T = \vdash$ using the standard-form encoding. Let $(P1, P2, \ldots, P10)$ be the standard-form encoding of $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), T)$ where $T = \vdash$. Now for each of the block apply the following $\mathsf{Rotation}_1$ operator. To get the encoding for $T = \dashv$ apply $\mathsf{Rotation}_2$ on each block of standard encoding of the tuple $(V, (r_L, c_L, b_L), (r_R, c_R, b_R), (r_B, c_B, b_B), \vdash)$.

Valid permutation of the standard form

Now we will give a set of valid permutations to the bits of the encoding of any element of $\bar{\Sigma}$. The set of valid permutations are classified into into 3 categories:

1. Part-permutation: The 10 parts can be permuted using any permutation from S_{10}
2. Block-permutation: In each of the part, the 4 blocks (say B_1, B_2, B_3, B_4) can be permuted is two ways. (B_1, B_2, B_3, B_4) can be send to one of the following
   a. Simple Block Swap: (B_3, B_4, B_1, B_2)
   b. Block Flip: $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$

The decoding function $\mathsf{Dec}_2 : \{0,1\}^{240\log n} \to \bar{\Sigma}$

We are now ready to describe the decoding function from $\Gamma = \{0,1\}^{240\log n}$ to $\bar{\Sigma}$.

- Identify the parts containing the encoding of V, r_L, c_L,b_L, r_R, c_R, b_R,r_B, c_B and b_B. This is possible because every part has a unique Hamming weight.
- For each part identify the blocks. This is also possible as in any part all the blocks have distinct Hamming weight. By seeing the positions of the blocks one can understand if flip was applied and to what and using that one case revert the blocks back to the standard-form.
- In the part containing the encoding of V consider the encoding-block. If the block is of the form $\{(\ell_1\ell_2\ell_3)\ \textit{such that}\ |\ell_1| = 2\log n, |\ell_2| = 2\log n - 1\ \textit{and}\ |\ell_3| \leq 2\log n - 2\}$ then $T = \{\vdash\}$. If the block is of the form $\{(\ell_3\ell_1\ell_2)\ \textit{such that}\ |\ell_1| = 2\log n, |\ell_2| = 2\log n - 1\ \textit{and}\ |\ell_3| \leq 2\log n - 2\}$ then $T = \{\top\}$. If the block is of the form $\{(\ell_2\ell_3\ell_1)\ \textit{such that}\ |\ell_1| = 2\log n, |\ell_2| = 2\log n - 1\ \textit{and}\ |\ell_3| \leq 2\log n - 2\}$ then $T = \{\dashv\}$.
- By seeing the encoding block we can decipher the original values and the pointers.
- If the $240\log n$ bit string doesn't have the form of a valid encoding, then decode it as $(0, \perp, \perp, \perp)$.

E.2.3 Proof of Transitivity of the function

To prove the function F_{74} is transitive we start with describing the transitive group for which the F_{74} is transitive.

The Transitive Group

We will now describe the group $\mathcal{T}$ acting on the cells of the matrix and a group $\widehat{\mathcal{T}}$ acting on the indices of a $6\log n$ bit string. First consider the group $\mathsf{Bt}_{2\log n}$ acting on the set $[2\log n]$. We will assume that $\log n$ is a power of 2. The matrix has rows $r_1, \ldots, r_n$, columns $c_1, \ldots, c_n$ and bricks $b_1, \ldots, b_n$. We have used the encoding function $\mathcal{E}'$ to encode the rows, columns and bricks. So the index of the rows, columns and bricks are encoded using a $6\log n$ bit string. From the Definition of $\mathcal{E}'$ and the group $\mathsf{Bt}_{2\log n}$ the following claim from Definition 81 and Claim 12.

▷ Claim 83. For any elementary permutation $\widehat{\sigma}$ in $\mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n}$ that takes $\mathcal{E}'(x)$ to $\mathcal{E}'(y)$ there exists a permutation $\sigma \in \mathsf{S}_n \times \mathsf{S}_n \times \mathsf{S}_n$ that takes x to y where $x, y \in (R \times \{0\} \times \{0\}) \cup (\{0\} \times C \times \{0\}) \cup (\{0\} \times \{0\} \times B)$. That is

$$\widehat{\sigma}[\mathcal{E}'(x)] = \mathcal{E}'(\sigma(x)).$$

Note that $\mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n}$ is not transitive on $6\log n$ bit string. Let us define the group $\widehat{\mathcal{T}} \subset \mathsf{S}_{6\log n}$ containing the following permutations:

1. all permutations in $\mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n}$,
2. $\mathsf{Rotation}_1$ to be applied on Block ($6\log n$-bit),

3. $\mathsf{Rotation}_2$ to be applied on Block ($6\log n$-bit).

$\widehat{\mathcal{T}}$ is the group generated by the above set of permutations and clearly $\widehat{\mathcal{T}}$ is transitive.

The group $\mathcal{T}$ will be the resulting group of permutations on the cells of the matrix induced by the group $\widehat{\mathcal{T}}$ acting on the $6\log n$ bit. The group $\mathcal{T}$ is generated by the following permutations:

1. all permutations induced by $\mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n} \times \mathsf{Bt}_{2\log n}$ that acts on the cell,
2. for all $i \in [n]$ row i is goes to column i, column i goes to brick i and brick i goes to row i. That is $\mathsf{Rotation}_1[\mathcal{E}'(r_i, c_i, b_i)] = \mathcal{E}'(\sigma(b_i, r_i, c_i))$ for all $i \in [n]$ where σ acts on the cell.
3. for all $i \in [n]$ row i is goes to brick i, column i goes to *row* i and brick i goes to column i. That is $\mathsf{Rotation}_2[\mathcal{E}'(r_i, c_i, b_i)] = \mathcal{E}'(\sigma(c_i, b_i, r_i))$ for all $i \in [n]$ where σ acts on the cell.

From the construction of $\widehat{\mathcal{T}}$ and $\mathcal{T}$ we have the following claim:

▷ Claim 84. For any permutation $\widehat{\sigma}$ in $\widehat{\mathcal{T}} \subset \mathsf{S}_{6\log n}$ acting on the $6\log n$ bit string there exists a permutation σ in $\mathcal{T} \subset \mathsf{S}_{3n}$ acting on the cell such that $\widehat{\sigma}[(\mathcal{E}'(x)] = \mathcal{E}'(\sigma(x))$ for all x in the domain of $\mathcal{E}'$.

▶ **Observation 85.** *One special set of permutation in $\widehat{\mathcal{T}}$ is called the "ith-bit-flips" or simply "bit-flips". For all $1 \le i \le 3\log n$ the ith-bit-flip applies the transposition $(2i-1, 2i)$ to the indices of the balanced-pointer-encoding.*

Since the $\mathcal{E}'$-encoding of the row $(r_k, 0, 0)$ uses the balanced binary representation of k in the first half and all zero sting in the second and third half, the j-th bit in the binary representation of k is stored in the $2j-1$ and $2j$-th bit in the $\mathcal{E}'$-encoding of r_i. So the j-th-bit-flip acts on the sets of rows by swapping all the rows with 1 in the j-th bit of their index with the corresponding rows with 0 in the j-th bit of their index. Also, if $i > \log n$ then there is no effect of the i-th-bit-flip operation on the set of rows.

One can make a similar observation for the columns and Bricks *also.*

From Definition 81, Claim 84 and Observation 85 we claim the following:

▷ Claim 86. The group $\mathcal{T}$ acting on the cells of of the matrix is a transitive group. That is, for all $1 \le i_1, j_1, i_2, j_2, k_1, k_2 \le n$ there is a permutation $\widehat{\sigma} \in \widehat{\mathcal{T}}$ such that $\widehat{\sigma}[\mathcal{E}'(i_1, 0, 0)] = \mathcal{E}'(i_2, 0, 0)$, $\widehat{\sigma}[\mathcal{E}'(0, j_1, 0)] = \mathcal{E}'(0, j_2, 0)$ and $\widehat{\sigma}[\mathcal{E}'(0, 0, k_1)] = \mathcal{E}'(0, 0, k_2)$. Or in other words, there is a $\sigma \in \mathcal{T}$ acting on the cell of the matrix that would take the cell corresponding to row r_{i_1}, column c_{j_1} and brick b_{k_1} to the cell corresponding to row r_{i_2}, column c_{j_2} and brick b_{k_2}.

From the Claim 86 we see the group $\mathcal{T}$ acting on the cells of of the matrix is a transitive. But it does not touch the contains within the cells of the matrix. But the input to the function F_{74} contains element of $\Gamma = \{0,1\}^{240\log n}$ in each cell. So we now need to extend the group $\mathcal{T}$ to a group G_2 that acts on all the indices of all the bits of the input to the function F_{74}.

Recall that the input to the function F_{74} is a $(n \times n^2)$-matrix with each cell of matrix containing a binary string of length $240\log n$ and for each of the cells the binary string contained in it has 10 *parts* of size $24\log n$ each and each part has 4 blocks of size $6\log n$ each. We classify the generating elements of the group G_2 into 4 categories:

1. Part-permutation: In each of the cells the 10 parts can be permuted using any permutation from S_{10}
2. Block-permutation: In block the 4 blocks (say B_1, B_2, B_3, B_4) can be permuted is two ways. (B_1, B_2, B_3, B_4) can be send to one of the following
   a. Simple Block Swap: (B_3, B_4, B_1, B_2)

b. Block Flip (#1): $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$
c. Block Flip (#2): $(\mathsf{flip}(B_1), \mathsf{flip}(B_2), B_4, B_3)$

3. Cell-permutation: for any $\sigma \in \mathcal{T}$ one has the do the following two steps simultaneously:
a. (Matrix-update) Permute the cells in the matrix according to the permutation σ. This does keeps each of the contains in the cells untouched - it just changes the location of the cells.
b. (Pointer-update) For each of blocks in each of the parts in each of the cells permute the indices of the $6 \log n$-bit strings according to σ, that is apply $\widehat{\sigma} \in \widehat{\mathcal{T}}$ corresponding to σ(existence of such permutation follows from Claim 84).

We now have the following theorem that would prove that the function F_{74} is transitive.

▶ **Theorem 87.** *The group* G_2 *is a transitive group.*

▶ **Theorem 88.** *The function* F_{74} *is a transitive function under the action of the transitive group* G_2.

Proof of Theorem 87. To prove that the group G_2 is transitive we have to show that for any indices $p, q \in [240n^2 \log n]$ there is a permutation $\sigma \in \mathsf{G}_2$ that would take p to q. So we can think of the index p as sitting in k_pth position ($1 \leq k_p \leq 6 \log n$) in the block B_p of the part P_p in the (r_p, c_p, b_p)-th cell of the matrix. Similarly, we can think of q as sitting in k_qth position ($1 \leq k_q \leq 6 \log n$) in the block B_q of the part P_q in the (r_q, c_q, b_q)-th cell of the matrix.

We will give a step by step technique in which permutations from G_2 can be applied to move p to q.

Step 1 **Get the positions in the block correct:** If $k_p \neq k_q$ then take a permutation $\widehat{\sigma}$ from $\widehat{\mathcal{T}}$ that takes k_p to k_q. Since $\widehat{\mathcal{T}}$ is transitive, such a permutation exists. Apply the cell-permutation from $\mathcal{T}$ corresponding to $\widehat{\sigma}$ on the cells. As a result the index p can be moved to a different cell in the matrix but, by the choice of $\widehat{\sigma}$ its position in the block in which it is will be k_q.
Without loss of generality, we assume the the cell location does not change.

Step 2 **Get the cell correct:** Using a series of "bit-flip" operations change r_p to r_q,c_p to c_q and b_p to b_q. Since one bit-flip operations basically changes one bit in the binary representation of the index of the row or column or brick such a series of operations can be made. This is another Cell-permutation. In this case we always do Matrix-update and Pointer-update simultaneously.
Since each bit-flip operation is executed by applying the bit-flips in each of the blocks so this might have once again changed the position of the index p in the block. But, even if the position in the block changes it must be a flip operation away. Or in other word, since in the beginning of this step $k_p = k_q$, so if k_q is even (or odd) then after the series bit-flip operations the position of p in the block is either k_q or $(k_q - 1)$ (or $(k_q + 1)$).

Step 3 **Align the Part:** Apply a suitable Part-permutation from S_{10} to ensure that the part P_p moves to part P_q.

Step 4 **Align the Block:** Using a suitable combination of Simple Block Swap and Block Flip ensure the Block number gets matched, that is B_p goes to B_q. In this case the cell or the Part does not change. But depending on whether the Block Flip operation is applied the position in the block can again change. But once again, the current position in the block k_p is at most one flip away from k_q.

Step 5 **Apply the final flip:** It might so happen that already we a done after the last step. If not we know that the current position in the block k_p is at most one flip away from k_q. So we apply the suitable Block-flip operation. Thus will not change the cell position, Part number, Block number and the position in the block will match. Hence we are done. ◀

Proof of Theorem 88. To prove that the function F_{74} is transitive we prove that for any elementary operations σ from the group G_2 and for any input Γ^{n^3} the function value does not change even if after the input is acted upon by the permutation σ.

Case 1: σ is a Part-permutation and **Case 2: σ is a Block-permutation** directly follows from our construction of the encoding so that Dec_2 is invariant under Part permutation and Block-permutation.

Case 3: σ is a Cell-permutation From Observation 23 it is enough to prove that when we permute the cells of the matrix we update the points in the cells accordingly.

Let $\sigma \in \mathcal{T}$ be a permutation that permutes only the rows of the matrix. By Claim 84, there exist a permutation $\widehat{\sigma} \in \widehat{\mathcal{T}}$ to be applied on every block and so that the contents of the cells will be updated accordingly. Similarly if σ only permute the columns or *Bricks* of the matrix we will be fine.

Finally, if $\mathsf{Rotation}_1$ was applied then for all i row i is swapped with column i, column i is swapped with brick i and brick i is swapped with row i then it is easy to see that $\widehat{\sigma}[\mathcal{E}'(i,0,0)] = \mathcal{E}'(0,i,0)$, $\widehat{\sigma}[\mathcal{E}'(0,i,0)] = \mathcal{E}'(0,0,i)$ and $\widehat{\sigma}[\mathcal{E}'(0,0,i)] = \mathcal{E}'(i,0,0)$. In that case the encoding block of the value part in a cell also gets swapped. This will thus be encoding the T value as $\top$. And so the function value will not be affected as the $T = \top$ will ensure that one should apply the $\mathsf{Rotation}_1$ on the cells of the input before evaluating the function. For $\mathsf{Rotation}_2$ we can argue similarly. Hence, F_{74} is transitive. ◀

E.2.4 Properties of the Function

▷ Claim 89. Zero error randomized query complexity of F_{74} is $\Omega(n^3)$.

Proof. The function $\mathsf{ModA3}*_{(n,n,n^2)}$ is a "harder" function than $\mathsf{A3}_{(n,n,n^2)}$. So $\mathsf{R}_0(\mathsf{ModA3}*_{(n,n,n^2)})$ is at least that of $\mathsf{R}_0(\mathsf{A3}_{(n,n,n^2)})$.

Now since, F_{74} is $\left(\mathsf{ModA3}*_{(n,n,n^2)} \circ \mathsf{Dec}_2\right)$ so clearly the $\mathsf{R}_0(F_{74})$ is at least $\mathsf{R}_0(\mathsf{A3}_{(n,n,n^2)})$. From [5] we have $\mathsf{D}(\mathsf{A3}_{(n,n,n^2)})$ is $\Omega(n^3)$. Hence $\mathsf{R}_0(F_{74})$ at least $\Omega(n^3)$. ◀

The following Claim90 that follows from the definition of the function $\mathsf{ModA3}*_{(n,n,n^2)}$.

▷ Claim 90. The following are some properties of the function $\mathsf{ModA3}*_{(n,n,n^2)}$:
$\mathsf{Q}(\mathsf{ModA3}*_{(n,n,n^2)}) \leq 3\mathsf{Q}(\mathsf{A3}_{(n,n,n^2)}) + O(n^2 \log n)$

Finally, from Theorem 44 we see that $\mathsf{Q}(F_{74})$ is at most $O(\mathsf{Q}(\mathsf{ModA3}*_{(n,n,n^2)} \cdot \log n)$. So combining this fact with Claim 89, Claim 90 and Theorem 67 (from [5]) we have Theorem 74.

E.3 Transitive function for Theorem 75

In [5] they have proved that $\mathsf{A3}_{(1,n,n^2)}$ is such that $\mathsf{R}(\mathsf{A3}_{(1,n,n^2)}) = \Omega(\mathsf{Q}_\mathsf{E}(\mathsf{A3}_{(1,n,n^2)})^{1.5}$. In this function $\mathsf{A3}_{(1,n,n^2)}$ number of *marked column* is 1, which is different from $\mathsf{A3}_{(n,n,n^2)}$. But for both of these function the domain is same. So, proof follows directly.

For transitive function first we will define $\mathsf{ModA3}*_{(1,n,n^2)}$ on $\bar{\Sigma}^{n \times n^2}$ similar as $\mathsf{ModA3}*_{(1,n,n^2)}$ in Section E.2. Then define a composition function with outer function $\mathsf{ModA3}*_{(1,n,n^2)}$ and

inner function Dec_2 as defined in Section E.2. Our final function is $F_{75} : \Gamma^{n^3} \to \{0,1\}$ such that $F_{75} = \mathsf{ModA3}*_{(1,n,n^2)} \circ \mathsf{Dec}_2$ and Dec_2 is a function from Γ to $\bar{\Sigma}$ where $\Gamma = 240 \log n$.

From the proof of Theorem 88 it directly follows that F_{75} is transitive under the action of the group G_2.

Since F_{75} is transitive, considering the fact that $\mathsf{ModA3}*_{(1,n,n^2)}$ is harder than $\mathsf{A3}_{(1,n,n^2)}$ and combining [5] result with Theorem 44 we have Theorem 75.

F Separation between sensitivity and randomized query complexity

[8] showed that functions that witness a gap between deterministic query complexity (or randomized query complexity), and $\mathsf{UC}_{\mathsf{min}}$ can be transformed to give functions that witness separation between deterministic query complexity (or randomized query complexity) and sensitivity. We observe that transformation the [8] described preserves transitivity. Our transitive functions from Theorem 73 along with the transformation from [8] gives the following theorem.

▶ **Theorem 91.** *There exists transitive functions F_{91} such that $\mathsf{R}(F_{91}) = \widetilde{\Omega}(\mathsf{s}(F_{91})^3)$.*

We start by defining *desensitisation transform* of Boolean functions as defined in [8].

▶ **Definition 92** (Desensitized Transformation)**.** *Let $f : \{0,1\}^n \to \{0,1\}$. Let U be a collection of unambiguous 1-certificates for f, each of size at most $\mathsf{UC}_1(f)$. For each $x \in f^{-1}(1)$, let $p_x \in U$ be the unique certificate in U consistent with x. The desensitized version of f is the function $f_{\mathsf{DT}} : \{0,1\}^{3n} \to \{0,1\}$ defined by $f_{\mathsf{DT}}(x_1x_2x_3) = 1$ if and only if $f(x_1) = f(x_2) = f(x_3) = 1$ and $p_{x_1} = p_{x_2} = p_{x_3}$.*

▶ **Observation 93.** *If $f : \{0,1\}^n \to \{0,1\}$ is transitive, then $f_{\mathsf{DT}} : \{0,1\}^{3n} \to \{0,1\}$ is also transitive.*

Proof. Let $T_f \subseteq \mathsf{S}_n$ be the transitive group corresponding to f and let $x_1x_2x_3 \in \{0,1\}^{3n}$ be the input to f_{DT}. Consider the following permutations acting on the input $x_1x_2x_3$ to f_{DT}:

1. S_3 acting on the indices $\{1,2,3\}$ and
2. $\{(\sigma,\sigma,\sigma) \in \mathsf{S}_{3n} | \sigma \in T_f\}$ acting on (x_1, x_2, x_3).

Observe that the above permutations act transitively on the inputs to f_{DT}. Also from the definition of f_{DT} the value of the function f_{DT} is invariant under these permutations. ◀

Next, we need the following theorem from [8]. The theorem is true for more general complexity measures. We refer the reader to [8] for a more general statement.

▶ **Theorem 94** ([8])**.** *For any $k \in \mathbb{R}^+$, if there is a family of function with $\mathsf{D}(f) = \widetilde{\Omega}(\mathsf{UC}_{min}(f)^{1+k})$, then for the family of functions defined by $\widetilde{f} = \mathsf{OR}_{3\mathsf{UC}_{\mathsf{min}}(f)} \circ f_{\mathsf{DT}}$ satisfies $\mathsf{D}(\widetilde{f}) = \widetilde{\Omega}(s(\widetilde{f})^{2+k})$. Also, if we replace $\mathsf{D}(f)$ by $\mathsf{R}(f)$, $\mathsf{Q}(f)$ or $\mathsf{C}(f)$, we will get the same result.*

Proof of Theorem 91. Let us begin with the transitive functions F_{73} from Section E.1 which will desensitize to get the desired claim. From Claim 77 and Observation 68 we have $\mathsf{R}(F_{73}) \geq \widetilde{\Omega}(\mathsf{UC}_{min}(F_{73})^2)$.

Let F_{91} be the desensitize F_{73}. From Theorem 94, Observation 93 we have the theorem. ◀

G Separation between quantum query complexity and certificate complexity

[2] constructed functions that demonstrated quadratic separation between quantum query complexity and certificate complexity. Their function was not transitive. We modify their function to obtain a transitive function that gives similar separation.

▶ **Theorem 95.** *There exists a transitive function* $F_{95} : \{0,1\}^N \to \{0,1\}$ *such that*

$$\mathsf{Q}(F_{95}) = \widetilde{\Omega}(\mathsf{C}(F_{95})^2).$$

We start this section by constructing an encoding scheme for the inputs to k-sum function such that the resulting function ENC-k-Sum is transitive. We then, similar to [2], define $\mathsf{ENC} - \mathsf{Block\text{-}k\text{-}Sum}$ function. Composing ENC-k-Sum with $\mathsf{ENC} - \mathsf{Block\text{-}k\text{-}Sum}$ as outer function gives us F_{95}.

G.1 Function definition

Recall that, from Definition 61, for $\Sigma = [n^k]$, the function $\mathsf{k\text{-}sum} : \Sigma^n \to \{0,1\}$ is defined as follows: on input $x_1, x_2, \ldots, x_n \in \Sigma$, if there exists k element $x_{i_1}, \ldots, x_{i_k}$, $i_1, \ldots, i_k \in [n]$, that sums to 0 (mod $|\Sigma|$) then output 1, otherwise output 0. We first define an encoding scheme for Σ.

G.1.1 Encoding scheme

Similar to Section 4.1.1 we first define the standard form of the encoding of $x \in \Sigma$ and then extend it by action of suitable group action to define all encodings that represent $x \in \Sigma$ where Σ is of size n^k for some $k \in \mathbb{N}$.

Fix some $x \in \Sigma$ and let $x = x_1 x_2 \ldots x_{k \log n}$ be the binary representation of x. The standard form of encoding of x is defined as follows: For all $i \in [k \log n]$ we encode x_i with with $4(k \log n + 2)$ bit Boolean string satisfying the following three conditions:

1. $x_i = x_{i1}x_{i2}x_{i3}x_{i4}$ where each x_{ij}, for $j \in [4]$, is a $(k \log n + 2)$ bit string,
2. if $x_i = 1$ then $|x_{i1}| = 1, |x_{i2}| = 0, |x_{i3}| = 2, |x_{i4}| = i + 2$, and
3. if $x_i = 0$ then $|x_{i1}| = 0, |x_{i2}| = 1, |x_{i3}| = 2, |x_{i4}| = i + 2$.

Having defined the standard form, other valid encodings of $x_i = (x_{i1}x_{i2}x_{i3}x_{i4})$ are obtained by the action of permutations $(12)(34), (13)(24) \in \mathsf{S}_4$ on the indices $\{i1, i2, i3, i4\}$. Finally if $x = \{(x_{ij}|i \in [k \log n], j \in [4]\}$, then $\{(x_{\sigma(i)\gamma(j)})|\sigma \in \mathsf{S}_{k \log n}, \gamma \in T \subset \mathsf{S}_4\}$ is the set of all valid encoding for $x \in \Sigma$.

The decoding scheme follows directly from the encoding scheme. Given $y \in \{0,1\}^{k \log n(4k \log n+8)}$, first break y into $k \log n$ blocks each of size $4k \log n + 8$ bits. If each block is a valid encoding then output the decoded string else output that y is not a valid encoding for any element from Σ.

G.1.2 Definition of the encoded function

ENC-k-Sum is a Boolean function that defined on n-bit as follows: Split the n-bit input into block of size $4k \log n(k \log n + 2)$. We say such block is a valid block iff it follows the encoding scheme in Section G.1.1 i.e. represents a number from the alphabet Σ. The output value of the function is 1 iff there exists k valid block such that the number represented by the block in Σ sums to 0 (mod $|\Sigma|$).

ENC − Block-k-Sum is a special case of the ENC-k-Sum function. We define it next. ENC − Block-k-Sum is a Boolean function that defined on n-bit as follows: The input string is splits into block of size $4k\log n(k\log n+2)$ and we say such block is a valid block iff it follows the encoding scheme of Section G.1.1 i.e. represents a number from the alphabet Σ. The output value of the function is 1 iff there exists k valid block such that the number represented by the block in Σ sums to $0(\text{mod } |\Sigma|)$ and the number of 1 in the other block is at least $6\times(k\log n)$. Finally, similar to [2] define $F_{95}:\{0,1\}^{n^2}\to\{0,1\}$ to be ENC − Block-k-Sum ∘ ENC-k-Sum, with $k=\log n$.

The proof of Theorem 95 is same as that of [2]. We give the proof here for completeness.

Proof of Theorem 95. We first show that the certificate complexity of F_{95} is $\mathsf{O}(4nk^2\log n(k\log n+2)$. For this we show that every input to ENC − Block-k-Sum, the outer function of F_{95}, has a certificate with $\widetilde{O}(k\times(4k\log n(k\log n+2)))$ many 0's and $O(n)$ many 1's. Also the inner function of F_{95}, i.e. ENC-k-Sum, has 1-certificate of size $\mathsf{O}(4k^2\log n(k\log n+2)$ and 0-certificate of size $\mathsf{O}(n)$. Hence, the F_{95} function has certificate of size $\mathsf{O}(4nk^2\log n(k\log n+2))$.

Every 1-input of ENC − Block-k-Sum has k valid encoded blocks such that the number represented by them sums to $(0 \text{ mod } |\Sigma|)$. This can be certified using at most $\widetilde{\mathsf{O}}(k\times(4k\log n(k\log n+2)))$ number of 0's and all the 1's from every other block.

There are two types of 0-inputs of ENC − Block-k-Sum. First type of 0-input has at least one block in which number of 1 is less than $6\times(k\log n)$ and the zeros of that block is a 0-certificate of size $\widetilde{O}(4k\log n(k\log n+2))$. The other type of 0-input is such that every block contains at least $6\times(k\log n)$ number of 1's. This type of 0-input can be certified by providing all the 1's in every block, which is at most $\mathsf{O}(n)$. This is because using all the 1's we can certify that even if the blocks were valid, no k-blocks of them is such that the number represented by them sums to 0 $(\text{mod } |\Sigma|)$.

Next, we prove $\Omega(n^2)$ lower bound on quantum query complexity of F_{95}. From Theorem 45, $\mathsf{Q}(F_{95})=\Omega(\mathsf{Q}(\text{ENC}-\text{Block-k-Sum})\mathsf{Q}(\text{ENC-k-Sum}))$. Since ENC-k-Sum reduces to ENC − Block-k-Sum, from Theorem 62 the quantum query complexity of the ENC − Block-k-Sum function is $\Omega\left(\frac{n^{\frac{k}{k+1}}}{k^{\frac{3}{2}}\log n(k\log n+2)}\right)$. Thus

$$\mathsf{Q}(F_{95})=\Omega\left(\frac{n^{\frac{2k}{k+1}}}{k^3\log n(k\log n+2)}\right).$$

Hence, $\mathsf{Q}(F_{95})=\widetilde{\Omega}(n^2)$, taking $k=\log n$. ◀

H Remaining cells of Table 1

In this section we prove that all function in white and yellow cell of Table 1 are transitive. Since PARITY $(\oplus)$ and AND $(\wedge)$ are symmetric they are also transitive by definition. The other functions that appear in the table are those in Definition 57 (NW^d for finite $d\in\mathbb{N}$), Definition 55 (RUB), Definition 59 (GSS_1^d) and Definition 60 (GSS_2).

We have the following corollary to the above observation:

▶ **Observation 96.** *The following functions are transitive:*

1. $\mathsf{OR}_n\circ\mathsf{AND}_n$
2. *$\widetilde{\wedge}$-tree for depth d for finite $d\in\mathbb{N}$,*
3. NW^d,
4. RUB,
5. GSS_1^d, *and*

6. GSS_2.

Proof. The transitivity of the functions either follows directly from their definitions or from the fact that they are composition of two transitive functions and hence the function is transitive using Observation 46. ◀

I Challenges with making transitive versions of "cheat sheet" based functions

In this section we show that it is not possible to give a quadratic separation between degree and quantum query complexity for transitive functions by modifying the cheat sheet function using the techniques in [2] which go via unambiguous certificate complexity.

Let us start by recalling the cheat sheet framework from [2]. Let $f : \{0,1\}^n \to \{0,1\}$ be a total Boolean function. Let $C(f)$ be its certificate complexity and $Q(f)$ be its bounded-error quantum query complexity. We consider the following cheat sheet function (also see Definition 63 for a formal definition), which we denote by $f_{CS,t} : \{0,1\}^{n\times\log t+t\times\log t\times C(f)\times\log n} \to \{0,1\}$:

- There are $\log t$ copies of f on disjoint sets on inputs denoted by $f_1, \ldots, f_{\log t}$.
- There are t cheat sheets: each cheat sheet is a block of $(\log t \times C(f) \times \log n)$ many bits
- Let $x_1, \ldots, x_{\log t} \in \{0,1\}^n$ denote the input to the $\log t$ copies of f and let $Y_1, \ldots, Y_t$ denote the t cheat sheets.
- Let $\ell = (f(x_1), \ldots, f(x_{\log t}))$. $f_{CS,t}$ evaluates to 1 if and only if Y_ℓ is a valid cheat sheet (see Definition 63)

[2] showed separations between several complexity measures using the cheat sheet framework. In [2], the separations that lower bound bounded-error quantum query complexity in terms of other complexity measures, for example degree, are obtained as follows:

1. Start with a total function $f : \{0,1\}^n \to \{0,1\}$ that has quadratic separation between quantum query complexity and certificate complexity: $Q(f) = \widetilde{\Omega}(n)$ and $C(f) = \widetilde{O}(\sqrt{n})$. Consider the cheat sheet version of this function $f_{CS,t}$, with $t = n^{10}$.
2. Lower bound $Q(f_{CS,t})$, for $t = n^{10}$ by $Q(f)$. This uses the hybrid method ([9]) and strong direct product theorem ([21].
3. Upper bound degree of $f_{CS,t}$ by using the upper bound on the unambiguous certificate complexity of $f_{CS,t}$.

Instead of degree, one might use approximate degree in the second step above for a suitable choice of f (see [2] for details).

A natural approach to obtain a transitive function with gap between a pair of complexity measures is to modify the cheat sheet framework to make it transitive. One possible modification is to allow a poly-logarithmic blowup in the input size of the resulting transitive function while preserving complexity measures of the cheat sheet function that are of interest (upto poly logarithmic factors). We show, however, that it is not possible to show a quadratic separation between degree and quantum query complexity for transitive functions by modifying the cheat sheet function using the techniques in [2] which go via unambiguous certificate complexity. The reason for this is that the unambiguous certificate complexity of a transitive cheat sheet function on N-bits is $\Omega(\sqrt{N})$ (see Observation 97) whereas the in Lemma 98 we show that the quantum query complexity of such a function is $o(N)$.

Note that this does not mean that cheat sheet framework can not be made transitive to show such a quadratic gap. If the cheat sheet version of a function that is being made

transitive has a better degree upper bound than that given by unambiguous certificate complexity then a better gap might be possible.

We now formalize the above discussion. First, we need the following observation that lower bounds the certificate complexity of any transitive function.

▶ **Observation 97** ([32]). *Let $f : \{0,1\}^N \to \{0,1\}$ be a transitive function, then $C(f) \geq \sqrt{N}$.*

The cheat sheet version of f, $f_{CS,t}$, is a function on $\widetilde{\Theta}(n + C(f)t)$ many variables, where t is polynomial in n. From the cheat sheet property the unambiguous certificate complexity of $f_{CS,t}$, denoted by $\mathsf{UC}(f_{CS,t})$, is $\widetilde{\Theta}(C(f))$. In Section I we show that $Q(f_{CS,t})$ is at most $\widetilde{O}(C(f)\sqrt{t})$. Thus in order to achieve quadratic separation between UC and Q, t has to be at least $\widetilde{\Omega}(C(f)^2)$.

Let $\widetilde{f_{CS,t}}$ be a modified transitive version of $f_{CS,t}$ that preserves the quantum query complexity and certificate complexity of $f_{CS,t}$ upto poly logarithmic factors, respectively. From Observation 97 it follows that $\mathsf{UC}(\widetilde{f_{CS,t}}) = \widetilde{\Omega}(\sqrt{n + C(f)t}) = \widetilde{\Omega}(C(f)^{3/2})$. On the other hand, since $\widetilde{f_{CS,t}}$ preserves the certificate complexity upto poly logarithmic factors, $\mathsf{UC}(\widetilde{f_{CS,t}}) = \widetilde{O}(C(f))$.

Upper bound on quantum query complexity of cheat sheet function

We use quantum amplitude amplification ([10]) in this section. Given a classical or quantum algorithm with success probability p, quantum amplitude amplification amplifies the success probability to $2/3$ (or some constant greater than $1/2$) by repeating the original algorithm $O(1/\sqrt{p})$ many times. Classically, such amplification requires $\Omega(1/p)$ repetitions (this is also sufficient).

▶ **Lemma 98.** *The quantum query complexity of $f_{CS,t}$ is $O(\sqrt{t} \times \log t \times \sqrt{n} \times \log n)$.*

Proof. Consider the following randomized query algorithm for $f_{CS,t}$:

- Choose $i \in [t]$ uniformly at random
- Check the ith cheat sheet: if the certificates in ith cheat sheet evaluate to i then proceed, otherwise return 1 and stop.
- Query the inputs of the functions to verify certificates. If the certificates match then return -1, otherwise return 1.

If the cheat sheet criteria is not satisfied, then the above algorithm makes no error. On the other hand, if the cheat sheet criteria is satisfied then there exists $i \in [t]$ that returns -1 when chosen in the first step of the above algorithm. Thus, the probability of algorithm being correct is at least $1/t$. Also, the number of queries made by the above algorithm is $\log t \times \sqrt{n} \times \log n$.

Thus, by amplitude amplification, $O(\sqrt{t})$ repetitions of the above algorithm gives us a bounded-error quantum query algorithm which makes $O(\sqrt{t} \times \log t \times \sqrt{n} \times \log n)$ many queries. ◀

We end this section by giving a concrete approach towards showing separation between unambiguous certificate complexity and quantum query complexity for a transitive functions using the cheat sheet method. We believe the it is possible to start with $f_{CS,t}$, for transitive function f and $t = \sqrt{n}$ and convert it to a transitive function that preserves the unambiguous certificate complexity and quantum query complexity upto poly logarithmic factors, while incurring a poly logarithmic blowup in the input size. However, we do not know how to prove quantum query complexity lower bound matching our upper bound from Lemma 98 for $t = \sqrt{n}$. We make the following conjecture towards this end.

▶ **Conjecture 99.** *There exists a transitive function $f : \{0,1\}^n \to \{0,1\}$ with $C(f) = \widetilde{O}(\sqrt{n})$ and $Q(f) = \widetilde{\Omega}(n)$. Let $f_{CS,\sqrt{n}}$ be the cheat sheet version of f with $\sqrt{n}$ cheat sheets. Then*

$$Q(f_{CS,\sqrt{n}}) = \Omega(n^{3/4}).$$

If true, the above conjecture should implies that for a transitive function f, $Q(f) = \widetilde{\Omega}(\deg(f)^{4/3})$.

[2] showed the for the quantum query complexity of the cheat function $f_{CS,t}$, i.e. $Q(f_{CS,t})$, is lower bounded by $Q(f)$, when $t = n^{10}$. Their proof goes via he hybrid method ([9]) and strong direct product theorem ([21]. Is is interesting to find the the constant smallest c such that $Q(f_{CS,n^c}) = \Omega(Q(f))$. We know that such a c must be at least than 1 (from Lemma 98) and is at most 10 (from [2]). We state this formally beow:

▶ **Question 100.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a non-constant Boolean function and let f_{CS,n^c} be its cheat sheet version with n^c cheat sheets. What is the smallest c such that the following is true:*

$$Q(f_{CS,n^c}) = \Omega(Q(f)).$$